

UiO : **Department of Informatics**
University of Oslo

Scalable and Efficient Web Application Architectures

Thin-clients and SQL vs. Thick-clients and NoSQL

Michael K. Gunnulfsen
Master Thesis Spring 2013



Scalable and Efficient Web Application Architectures

Michael K. Gunnulfsen

Spring 2013

Abstract

Modern Web applications have lately seen an increase of popular Web 2.0 patterns, such as user-participation and rich user-interfaces. These applications require highly dynamic page generation techniques and flexible and efficient database solutions. At the same time, an increase in JavaScript engines in modern Web browsers has lead to the development of many new and exciting Web application architectures.

In this thesis, we investigate such a modern Web application architecture and compare it with a traditional architecture. We define the modern approach as thick-client JavaScript architectures that incorporate NoSQL database technologies, while the traditional approach is thin-client architectures that uses SQL database technologies.

The results were that the modern approach is both more scalable, efficient, in terms of response times, and the NoSQL solution is more flexible because of its simplified programming model. However, there are issues with the modern approach, in which some principles from the traditional architecture should be preserved.

Contents

I	Introduction	1
1	The Thesis at a Glance	3
1.1	Motivation	3
1.2	Goals	4
1.3	Problem Statement	5
1.4	Approach	6
1.5	Proposed Solution	6
1.6	Evaluation	7
1.7	Work Done	8
1.8	Results	8
2	Background	11
2.1	Introduction	11
2.2	From Web Sites to Web Apps	11
2.2.1	History of The World Wide Web	11
2.2.2	The Early Days	12
2.2.3	Modern Web Applications	13
2.3	Web Technologies	14
2.3.1	Web and Application Servers	14
2.3.2	The Web Browser	15
2.3.3	JavaScript	17
2.3.4	Client-server Interaction Schemes	19
2.3.5	HTTP Sessions	20
2.3.6	Representational State Transfer	20
2.3.7	JSON	21
2.3.8	Business Logic and View Logic	21
2.3.9	HTML Template Rendering	22
2.3.10	Databases	22
2.4	Summary	25
3	Design Alternatives for Modern Web Applications	27
3.1	Introduction	27
3.2	Reference-model 1.0	28
3.2.1	The Three-Layered Architecture	28
3.2.2	The Front-End	31
3.2.3	Platform Environment	32
3.2.4	Examples of traditional Web architectures	32

3.3	Reference-model 2.0	35
3.3.1	Front-end Frameworks	36
3.3.2	Thick-Client Concepts	36
3.3.3	The Simplified Back-end	38
3.3.4	Modular JavaScript	39
3.3.5	Client-side Page Rendering	40
3.3.6	Client State and Navigation Handling	41
3.3.7	Alternative Front-end Design Patterns	41
3.4	The Solutions Chosen	42
3.5	Summary	43
II	The Project	45
4	Shredhub, a Web 2.0 Application	47
4.1	The Concept	47
4.2	User Functionality	48
4.3	Pages and User Stories	48
5	Architecture 1.0	55
5.1	Introduction	55
5.2	Architectural Overview	55
5.3	The Presentation Layer	56
5.3.1	Authentication	56
5.3.2	State and Session Management	57
5.3.3	Input Validation	58
5.3.4	Controllers	58
5.3.5	Views	59
5.3.6	Summary of The Presentation Layer	61
5.4	The Domain Logic Layer	61
5.4.1	Service Functions and Domain Objects	62
5.4.2	Summary of The Domain Logic Layer	63
5.5	The Data Source Layer	63
5.5.1	SQL Implementation	64
5.5.2	How Much Data to Fetch	66
5.6	Summary of The Data Source Layer	67
5.7	Summary	67
6	Architecture 2.0	69
6.1	Introduction	69
6.2	Architectural Overview	69
6.3	The Front-end	70
6.3.1	The Bootstrapping Process	70
6.3.2	Router	72
6.3.3	Models	73
6.3.4	Collections	73
6.3.5	Views	73
6.3.6	The Mediator	76

6.3.7	Session	76
6.3.8	Summary of The Front-end	77
6.4	The Back-end	78
6.4.1	The Rest API	78
6.4.2	The Data Repository Layer	79
6.4.3	Authentication	80
6.4.4	The Databases	80
6.4.5	Summary of The Back-end	84
6.5	Summary	84
7	Performance and Source Code Analysis	87
7.1	Introduction	87
7.2	Web Application Performance	88
7.3	Hardware and Software Used for Testing	88
7.4	Performance and Scalability Tests	89
7.4.1	Test 1 - Page Loading Tests	89
7.4.2	Test 2 - Interactive User-Action Tests	92
7.4.3	Test 3 - Back-end Scalability Test	96
7.4.4	Test 4 - Database Performance	99
7.4.5	Test 5 - Code Flexibility Test	99
7.5	Summary	102
7.5.1	Performance Results	102
7.5.2	Scalability Results	102
7.5.3	Code Quality Results	102
III	Discussion and Conclusion	105
8	Discussion	107
8.1	Introduction	107
8.2	Page Rendering	107
8.3	State and Business Logic on the Client	108
8.4	NoSQL and SQL Implementation	111
8.5	Strengths and Limitations of the Study	112
8.5.1	Srenghts	112
8.5.2	Limitations	113
8.6	Related Work	113
8.7	Implications on Practice	114
8.8	Summary	115
9	Conclusion	117
IV	Appendix	119
10	Appendix A	121
10.1	Architecture 1.0 - Presentation Layer Example	121
10.2	Architecture 1.0 - View Logic Example	122
10.3	Architecture 1.0 - Domain Logic Layer Example	124

10.4	Architecture 1.0 - Add Dig User-feature	125
11	Appendix B	131
11.1	Architecture 2.0 - Domain Logic Example	131
11.2	Architecture 2.0 - View Logic Example	133
11.3	Architecture 2.0 - Add Dig User-feature	135

Preface

I am very thankful to Eric Bartley Jul, my supervisor, for all our educational meetings. Eric has so many exciting stories to tell, and so much knowledge to convey. I want to thank Eric for academic education, thesis guidance, and his truly positive and encouraging attitude.

I also want to thank my friends and family for support and guidance. Especially my mother, Trine, for research advice.

Part I

Introduction

Chapter 1

The Thesis at a Glance

1.1 Motivation

Web 2.0 is a popular term for the second generation World Wide Web. A new paradigm has emerged with the Internet's changing usage pattern that is increasingly becoming more social[94]. In typical Web 2.0 sites, the users have their own profile account, they connect, collaborate and share information. What is special with these applications is that each user has a personalized view of the Web page, based on their account information and connections to other users. Many modern Web applications incorporate such social networking features.

A characteristic property with Web 2.0 applications is that the users provide and consume the majority of the content. This often leads to large quantities of stored data. In addition, Web 2.0 applications often offer highly rich and interactive user-interfaces [128, p. 158]. The Web pages contain graphical widgets that display animated behavior, and clickable components that generate interactive responses. For this particular reason, such interactive Web sites are often referred to as Web applications, or in short "Web apps" rather than Web pages[131].

A typical Web 2.0 application brings challenging requirements to its software architecture. Common usage behavior normally requires many and frequent data retrievals and updates, and the content should always be up-to-date. Considering the fact that the amount of persisted data is often very large, and the amount of simultaneous users is often very high, it is obvious that the back-end system must be both scalable and efficient. Also, the complex user interfaces require efficient graphical user interface code. This code has to be flexible and maintainable, in order to facilitate new user requirements, something that occurs very frequently for popular Web 2.0 sites[37].

It is not without reason that the quality of the user interfaces of modern Web applications has increased dramatically the last years [66]. Rich user-experiences primarily relies on efficient Web browsers that can execute client-side dynamic behavior. Earlier, demanding client-side behavior had to be implemented by technologies such as Flash[4] and Java applets [81, p. 4]. These were highly efficient technologies that could run inside certain

browsers. Older JavaScript engines, however, were not that efficient, so the purpose of the JavaScript was mostly limited to input validation and simple graphical behavior. However, modern JavaScript engines have become so powerful, that the browsers are now able to execute complex JavaScript code highly efficiently. This has facilitated the design of rich and interactive user-experiences that runs in any modern browser.

The traditional software architectures for Web applications has in the last decade followed a thin-client approach. These architectures are heavily server-oriented in which most of the source code is executed in an application that runs on a Web server (the back-end). Also the information content is often stored in a relational database. For every URL request, the back-end is responsible for creating and delivering a dynamically generated HTML page to the client. The HTML code delivered to the client often contains a set of independent and unstructured JavaScript functions that generate the necessary interactive behavior. However, the rise of Web 2.0 has brought some interesting technologies and architectural concepts that makes it possible to build pure JavaScript applications that run primarily in the browser. Previously, this used to be rather pointless, because some five years ago, browsers were not able to host large-scale JavaScript applications. Also, such JavaScript Web applications are challenging to implement because the language itself lacks common features such as classes and namespaces. However, modern JavaScript frameworks provide syntactic sugaring enhancements that improves the programming experience, which makes it easier to build thick-client Web applications in pure JavaScript. In addition, modern Web applications tend to incorporate database solutions other than SQL, with the purposes of achieving better scalability and a simplified programming model. Such solutions are commonly referred to as NoSQL[73].

This thesis investigates the pros and cons for developing a traditional Web 2.0 application by either using a traditional thin-client architecture backed by a SQL database, or an innovative thick-client architecture that uses NoSQL technologies.

1.2 Goals

The major goal for this thesis is to find optimal architectural principles in terms of scalability and performance, for building typical Web 2.0 applications. Performance in this case is scoped to end-user response times. The work that is done involves finding sustainable solutions for both the traditional thin-client and the thick-client approach. This way, a proper comparison will help identify significant pros and cons in each architecture. The results are used to give guidelines for good architectural decisions, and propositions for hybrid solutions and future work.

In the study of these two architectural approaches, there has also been a focus on code quality, considering this is important in order to facilitate future growth in any modern Web application. The purpose of this is to find sustainable methods to structure the code in both the thin-client and

thick-client approach. This way, the codebases can also be compared in order to give further reasons to choose any of the two architectures.

It is important to have in mind that a server back-end can only be scalable up to a certain point, in which case the only solution to achieve further scalability is to upgrade the server hardware, or add more physical servers. Considering that this is both resource demanding and time consuming, I limit my investigation to apply mostly to single server deployments. Additionally, as for the thick-client JavaScript architecture, the goal for it is to work ideally in modern Web browsers, as older browsers lack ability to execute large-scale JavaScript applications.

Also, even though security is a big issue when it comes to designing Web applications, I have decided not to focus on this, simply because it would be too time consuming and laborious. The only security issue I bring to attention is authentication, because it has a very high impact on the overall architecture of the Web application. The goal is not to find the best authentication protocol for any Web app architecture, but rather to find reliable solutions that fit the architectures we discuss.

1.3 Problem Statement

In this section, we look at the main questions we want to solve in the thesis. The term “Web app” is being used to refer to typical modern Web 2.0 applications that contain rich and responsive user interfaces, incorporates social networking, and manipulates large amounts of persisted data. The main question we ask concerns the benefits, if any, for having a **thick-client** Web app architecture instead of a traditional **thin-client** architecture. We separate this problem statement into three more specific questions:

1. In traditional Web apps, HTML pages are dynamically generated on the server, which is done in every URL request. Can Web apps achieve better performance and scalability results by dynamically creating pages on the client, and by using the server only as an interface to the database?
2. In traditional Web apps, state is kept on the server. Can Web apps perform and scale better by moving state completely to the client? Also, if state is moved to the client, can Web apps benefit from moving business logic operations to the client as well? Benefits are evaluated in terms of performance and also programmer satisfactory aspects such as code flexibility and simplicity.
3. In traditional Web apps, the data is persisted in a relational database management system by using SQL as the query language. However, new types of database systems (commonly called NoSQL) offer a different, schema-less persistency solution that is often specialized to fit a specific type of application. Is there any such database system that particularly suits the thick-client architecture, and can they make it perform better than a typical SQL database implementation?

Are there any programmer satisfactory advantages of using such a NoSQL database in Web apps?

1.4 Approach

Considering the main goals for this thesis is to compare two different approaches to Web application architecture, a good way to get sustainable and reliable results is to design and implement an actual application, by using both of these approaches. I have defined two manifests that outlines the major principles in respectively a traditional thin-client architecture that uses SQL, and an innovative thick-client architecture that uses NoSQL. These manifests are called *Reference-model 1.0* and *Reference-model 2.0*. In addition, I have come up with an idea for a typical Web 2.0 application that conforms to the user requirements stated earlier. This application is built twice from the start with two completely different architectures. One implements the principles from *Reference-model 1.0* while the other implements *Reference-model 2.0*.

To get good and valuable results, extensive system-testing has been done on both prototypes. A set of concrete test cases have been proposed and executed on both architectures. The test cases are designed to test the performance and scalability behavior of the applications, in addition to a minor test that studies the two source codebases. To be able to do this, a lot of dummy data has been generated and used to populate the databases. In addition, the testing was done by running simulations that generates a high number of simultaneous user requests.

1.5 Proposed Solution

For this thesis I have proposed a Web application called *Shredhub*, which is a social networking site primarily designed for musicians. The application features common behavior found in traditional Web 2.0 applications.

Shredhub has been built twice, using a traditional approach, named *Architecture 1.0*, and an innovative approach, named *Architecture 2.0*.

Architecture 1.0 conforms to the following principles:

- HTML is dynamically generated on the server
- State handling and business logic is implemented on the server
- Stores data in a SQL database
- A set of autonomous JavaScript functions are used to generate quick and responsive behavior

Architecture 2.0 conforms to the following principles:

- HTML is dynamically generated on the client
- State handling and business logic is implemented on the client
- Uses NoSQL technologies to persist data
- The application is kept in the browser and implemented purely in JavaScript

1.6 Evaluation

The evaluation is based on implementation observations and the tests that were performed on Architecture 1.0, and Architecture 2.0. The tests were designed to investigate efficiency, scalability and to some extent, source code quality.

Efficiency has been evaluated in terms of the response time when an action is performed on Shredhub. The action might be clicking a link in a tab that leads to a new page, uploading a video, rating a video, etc. The evaluation is based upon how fast the architecture is able to generate the result. In addition, to investigate database efficiency, evaluation is in this case based on how fast the architectures execute the most popular database queries used in Shredhub. Also, an evaluation of the back-end efficiency is based on how much time is spent on the server for each test case.

Scalability is evaluated in terms of how well the architectures deal with an increasing number of simultaneous requests. This has been done by creating multiple threads that simultaneously execute common user actions on Shredhub. The evaluation is based on, for each number of simultaneous requests in the set of $U=\{1,10,100,200,400,600,800,1000\}$ where U is simultaneous users, how fast the results are being delivered, and how many users the Web app can at most handle before it no longer returns valid answers.

Source code quality is only a minor evaluation point in this thesis, much due to the limited amount of time there was to test this in this thesis. However, considering that this is also very relevant in terms of comparing the two software architectures, some evaluation has been done. The two codebases are compared in terms of the amount of lines of source code, the number of different programming languages used, and lastly, how much code had to be modified and added when a new user requirement was introduced and was to be implemented in the already finished codebase. A final test case was designed to involve both the implementation of a graphical user interface component, a business logic operation, and a new database operation. In addition, an observation of general programmer satisfactory aspects was noted during development.

1.7 Work Done

The initial work done for this thesis was to identify common characteristics in modern Web 2.0 applications. This led to the design concept for a Web application that could incorporate these characteristics in the application's user requirements. At the same time, a lot of work has been done in studying architectural trends in modern Web applications. A lot of time was spent looking at open-source code repositories, reading technology blogs, books, watching Web-seminars and presentations, and reading online discussions on modern Web architecture. Coincidentally not a lot of research has been done on thick-client JavaScript based Web applications, therefore much of the knowledge is based on the sources just described. It was important to get a comprehensive overview of the common trends in Web architecture in order to decide on the most industry-relevant solutions for the prototypes that were developed in this project.

Further, the work involved the design and implementation of the two prototypes. The implementation process had a strong focus on keeping the applications looking exactly the same from a user's perspective, while at the same time focusing on developing the architectures in two completely different ways.

The last part involved deciding how the two architectures were to be tested and compared. This work involved defining a set of concrete test cases aimed to test the performance and scalability behavior for the applications. Finally the two architectures were deployed on a test machine, and the tests were executed on them. In addition, the two codebases were revisited in the implementation of an additional user feature, so that the code could be compared in terms of flexibility and simplicity.

1.8 Results

The results show clear advantages for Architecture 2.0. Modern Web 2.0 applications can successfully benefit from generating dynamic HTML on the client. The reason is that generating HTML on the server can be a tedious job, especially when the amount of simultaneous users is high. This behavior was fully possible to implement with client-side JavaScript. Also, state and business logic were successfully moved to the client by building a full-scale JavaScript application that is sent to the browser on initial Web page requests. The benefits are that a lot of load is taken off the server, leading to higher scalability and performance results. However, it does require a modern browser, and also the initial page request could be significantly slow. Also, another disadvantage was that some business logic had to be duplicated on both the client and the server, and also that the Web app itself is not properly picked up by Web crawlers. The programming benefits were great, because code that manipulates the user-interface lies closer to, and cooperates better with the application's logic code, because they exist under the same programming language.

abstraction (module).

As for the database solutions, there are very many NoSQL technologies to choose from. However, one solution was found for Architecture 2.0, that nicely fits the domain for Shredhub. The results were that some queries were very efficient in cases where the domain could fit under the same database entity, but slow in cases where multiple entities had to be joined together. In those cases, SQL were much more efficient. On the other hand, the NoSQL solution was very programmer satisfying, because no translation between objects in the back-end and the database had to be done, because they share the same programming language.

Chapter 2

Background

2.1 Introduction

Web applications have in the last few years seen a dramatic change in both behavior and magnitude. They have grown from being a collection of simple and static Web pages into highly dynamic, and interactive applications with rich user interfaces. Previously, interactive behavior in Web sites were usually performed by Java applets and Flash applications [81] that could run inside the browser. But as JavaScript engines and Web browsers have become significantly more powerful, such behavior is increasingly being implemented exclusively with JavaScript[81]. Together with this shift towards highly interactive Web applications, the user behavior is at the same time increasingly becoming more social. Users make up the main data content of Web applications by socially interacting with each other and adding content to the pages.

This chapter outlines recent trends in applications that can be found on the Internet; commonly named Web 2.0[128]. We look at the technologies that enable applications to run on the Internet, and more specifically, the software architectures and technologies that are commonly used for developing traditional Web 2.0 applications.

The chapter begins with a short history of the World Wide Web, then a discussion of how the Web has changed from being simple and static Web documents, into dynamic Web 2.0 applications. Then, we present an overview of the key attributes and common user behavior that is found in modern Web applications. Finally an overview of some software architectures and technologies that are commonly used to implement such applications is given. This background material will be the foundation of the study that has been done in this thesis.

2.2 From Web Sites to Web Apps

2.2.1 History of The World Wide Web

The World Wide Web (www) was first introduced by Sir Tim Berners Lee at the CERN research laboratory in 1989[123]. He laid out a proposal for

a way of managing information on the Internet through hypertext, which is the familiar point-and-click navigation system to browse Web pages by following links. At this time, Tim Berners Lee had developed all the tools necessary to browse the Internet. This included the HyperText Transfer Protocol (HTTP), which is the protocol used to request and receive Web pages. The HyperText Markup Language (HTML), which is a markup language that describes how information is to be structured on a Web page. The first Web server that could deliver Web pages, and he built a combined Web browser and editor that was able to interpret and display HTML pages. By 1993, CERN declared that the World Wide Web would be open for use by anyone[1]. This same year, the first widely known graphical browser was released under the name Mosaic[84], which would later become the popular Netscape browser. Later in 1995, Microsoft would release their compelling browser Internet Explorer[62], leading to the first "browser wars" where each competitor would try and add more features to the Web. Unfortunately, new features were often prioritized in favor for bug fixes, leading to unstable and unreliable browser behavior. Example outcomes were the Cascading Style Sheet (CSS) [32], which is a language that describes how the HTML elements should appear in the browser. And also, Netscape's JavaScript [22] was developed to add dynamic behavior that could run in the browser. Microsoft created a replicated version of JavaScript, which they named JScript[69].

2.2.2 The Early Days

In the mid 90's, Web sites were mostly **static**, meaning that the documents received from a Web server were exactly the same each time it was requested. This was only natural, as the majority of Web sites were pre-generated HTML pages with lots of static content, for example a company's, or a person's home page. Later, however, the need for user-input became apparent as applications like for example e-commerce sites would require two-way communication. User input was not part of the first version of HTML (1.0), which led to the development of HTML 2.0.¹ This standard included **Web forms**, which allowed users to enter data and make choices that were sent to the Web server. The development of Web sites grew into becoming **dynamic** Web pages. This means that the server responds with different content depending on the input received in HTTP requests. To enable this, there has to be a program running on the server that can evaluate the HTTP request, and generate a proper HTML page depending on the request itself, and the application's state. This is called *server-side dynamic page generation* [117, p.691]. Another common scenario is *client-side dynamic page generation*, in which a program is sent to the browser, and executed inside the browser. Examples are JavaScript, and applets, which are programs that are compiled to machine code on the client's machine and executed inside the browser. Because applets are compiled

¹At the time, HTML was being developed by the Internet Engineering Task Force (IETF)[119], an organization that develops and promotes Internet standards.

to machine code, they execute faster than JavaScript, and therefore, such technologies have for long been favored for implementing performance demanding behavior in the browser. Examples are Java applets, Microsoft's ActiveX[3], and Adobe's Flash[4].

The Problem with Client-side Technologies

Java applets and Flash had become popular choices for client-side dynamic page generation by the year 2000 [81, p.2-3], and they still exist in many Web applications. There are many problems with this approach however. For instance, a plugin is usually required for running such applications inside the browser, developers need to know an additional programming model, the user interface tends to look different than the rest of the HTML page, and on top of this there have been numerous examples of security violations with the technologies themselves [117, p.875-877]. Choosing JavaScript primarily for client-side interactivity would be a preferable solution, because it doesn't require an additional programming language or run-time environment considering JavaScript is already supported in all popular browsers. Unfortunately, this technology has also had its issues ever since it was introduced. Partly because of its buggy implementations due to the scurrying development processes in the early browser wars, which has led to different JavaScript interpreter implementations by the various browser vendors. But also because browsers have not had the ability to execute JavaScript fast enough to enable satisfying dynamic behavior. For this reason, JavaScript has for long been used as an add-on language for HTML to perform simple roll-over effects, input validation, pop-up windows, and the like.

However, a lot of work has been done to provide a standardization of the JavaScript programming language. And lately, browser vendors such as Google[51] and Mozilla[85] have improved the engines that execute JavaScript to enable the execution of performance demanding processing jobs.

2.2.3 Modern Web Applications

Recently, a lot of work has been done in standardizing Web technologies, such that applications can be built to run on all browser. Examples include the work on the newest version of HTML, CSS and JavaScript. This dramatically simplifies the development of dynamic and interactive client-side behavior and media incorporation without the need for additional plugins. The work on improving and standardizing JavaScript has made it the assembly language of the Web, and is now one of the most popular programming languages in the world [65]. With this trend towards client-side development, the Web has seen an expanding growth in applications with rich user interfaces and lots of interactive behavior, that looks almost like native running desktop applications. Such applications are often called simply "Web apps".

The Social Web

In addition to interactivity and responsive behavior, there is another trend that is increasingly becoming a key factor in modern Web apps; namely social interactions. Many modern Web apps base the information content that makes up the site on what the users add to the page. Usually this includes users posting blog posts, comments, images and other sorts of data information. And in addition, the users connect to each other in a "social network". Popular social network applications are Facebook[39], Twitter[125], Pinterest[97], and many others.

Applications that incorporate social networking features and let users add content naturally leads to large quantities of persisted data. In light of this, many new database management systems have lately been introduced to the Web industry, with the intent of achieving more scalable solutions. Such technologies often have in common that they don't follow the traditional relational database structure (I.e SQL-based), but adopts other less structural approaches. Such databases are commonly being referred to as NoSQL[73]. A big reason why they don't adhere to the traditional relational structure like SQL is that this technology has showed not be fairly suited to be distributed over multiple database servers[2]. Most NoSQL databases, on the other hand, has showed its ability to scale very well over multiple servers, making it a good choice for Web 2.0 applications that persists large data quantities. In addition many of these databases tend to fit a specific type of application, both in terms of performance, and a simplified programming model, making it easy to communicate with the database from the application.

A final note with the Web applications just described is that they often offer their services to external third party clients through, what's commonly called their "public API". This means that other external applications might use functionality that the application is offering publicly as a service, and incorporate this functionality into their own app. This is called a service-oriented architecture[36].

2.3 Web Technologies

Having looked at how the Web started, and given an overview of common user features for Web 2.0 applications, we will now focus on the technologies that host these applications. We will begin this section with an introduction to the client-side technologies that executes Web apps, then we discuss some common architectures and principles for designing them.

2.3.1 Web and Application Servers

Web servers, also called HTTP servers, is a program running on a dedicated server machine, that offers Web content to client users. The client is usually a Web browser, but it could also be a Web crawler, who often intends to gather information on Web pages for searching purposes. The Web server manages HTTP communication with the client users, and serves

static content like images, videos, or stylesheet files. Examples of popular Web servers are Apache Web Server[7], and Microsoft Internet Information Server[63]. The Web server is responsible for delegating requests for dynamic content to an **application server**. The application server hosts the Web application itself, which is often just called the back-end², and hides the low level implementation of HTTP, typically by wrapping HTTP-header info into separate programming language variables. The application server can route specific URL requests to appropriate handlers in the Web application. Examples of application servers are Apache Tomcat [10] for Java, and Rack [105] for Ruby. Application servers usually support one or more **Web application frameworks**, which simplifies the development of a Web application in a specific programming language. Examples are SpringMVC for Java [104], Ruby on Rails for Ruby[109], and Django for Python[34].

2.3.2 The Web Browser

Browsers are software applications that requests and displays content on the Internet. The information is usually expressed as HTML pages, but it can also be other types of data, for instance images, script files, PDF files, or videos. The way browsers should interpret Web content is specified by World Wide Web Consortium (W3C)[132], however up until recently, the various browser vendors have usually not completely conformed to the whole specification but instead developed customized solutions. This has caused many compatibility issues for Web developers.

High-level structure

The browser's software stack consist of a set of components that each has individual responsibilities, and cooperates with the work of fetching and displaying Web resources. The main components of a browser are:

1. User interface
2. Browser engine
3. Rendering engine
4. Networking
5. JavaScript interpreter
6. UI backend
7. Data persistence

The rendering engine is a very important part in the process of displaying a resource. Its responsibility is to get the document from the network layer, render the document and finally paint the result on the

²In conjunction to the back-end, the application's front-end concerns the code that runs in the browser.

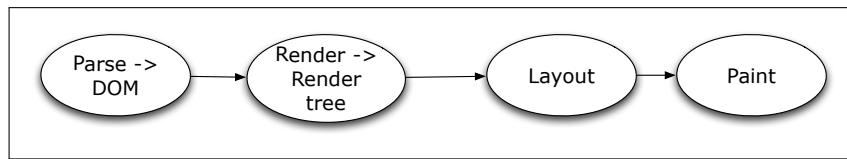


Figure 2.1: The rendering engine's responsibility

display. The process of rendering the document is showed in figure 2.1. Note that this process is iterative and will happen repetitively until the whole HTML page with all its external resources are completely processed. The rendering engine's lifetime is **single-threaded** and runs in an infinite loop that listens to events. An event might be to calculate a new position of an element, perform painting on new or modified HTML elements or handle a mouse click. However, if multiple external resources are to be fetched at the same time, the browser can, and often will create multiple HTTP connections that will run in parallel to efficiently load content that needs to be contained in the main HTML document.

WebKit[122] and Gecko[49] are two popular rendering engines that implements the rendering process in figure 2.1, however they do differ slightly in their internal behavior. WebKit is the engine that runs Chrome and Safari, while Gecko runs Firefox. In this section we limit the discussion to concern only these platforms, as they are built upon open source solutions and thus have available technical descriptions. The text that follows describes the process of rendering a complete HTML page. This usually happens when the client-browser requests a URL for an HTML page and the browser "refreshes" the page with the new content.

Parsing The rendering of an HTML page starts when the networking layer is instructed to fetch a URL, say *www.google.com*. Once the HTML page is fetched, the browser will immediately start fetching all external **links** that are contained inside it, from top to bottom. This could be links to CSS stylesheet pages, JavaScript pages, images, videos, etc. The rendering engine will continuously request chunks of HTML and CSS data from the networking layer, that it parses. An important feature of the HTML document's structure is that all the markup elements (tags) are nested in a hierarchical structure. Thus, when the HTML document is parsed, its tags are laid out in a tree structure.

Whenever the parser hits a script tag, it will first fetch the script if it is referencing an external file. Then, it will execute the script immediately. Unless the tag is marked "deferred" in which case the handling of the script will be postponed until the HTML parsing is done, the parser has to wait until the script is both fetched and executed. This is because the script might try to manipulate the HTML. To improve performance by avoiding the HTML parser to block while scripts are loaded, scripts can also be marked as "async" in which case the modern browser will generate a separate thread that fetches (if the script is not embedded in the HTML)

and parses the script, while the main parser thread can continue and parse HTML.

The tree that is generated is called the **DOM tree**, where each tree element is named a DOM element. The DOM tree offers a programming interface (API) that can be used by JavaScript in order to manipulate the HTML. When the whole HTML page is completely parsed, the rendering engine will start executing the scripts that are marked "deferred". When these scripts are finished executing, the browser will generate a **DOMContentLoaded** event. Finally, when all the external resources are fetched and parsed, the browser will generate an event called the **load event**. The purpose of these events is that JavaScript execution can be set to execute, first once any of them is triggered.

Rendering While the DOM tree is being populated, the rendering engine will also start generating the **Render tree**. This is another tree that is a visual representation of the DOM tree, and in effect decides the style and order of how the DOM elements should be laid out. Every element in the Render tree has a reference to its DOM node, a style, and in addition they know how to layout and paint itself and its children. In effect each render element represents a visual rectangle on the screen.

Layout and Painting In the layout process, each render element is given coordinate instructions for where on the screen it will be placed, and its size. The calculation is performed recursively from the root HTML node to the bottom. The painting process does the actual work of painting the elements on the screen. It does so by iterating through the rendering tree and paints each component.

2.3.3 JavaScript

JavaScript is an interpreted programming language primarily built for manipulating Web pages. The language was developed at Netscape in 1995, during the time when Netscape and Microsoft were battling for the majority of browser users. The language itself was built in only 10 days, by Brendan Eich[17]. He had instructions to develop a language that would look like Sun's Java, only simpler, interpreted and easy to integrate into Web pages, so that it would appeal to non-professional developers. Eich designed the language to follow much of the syntax from the C programming language, only simpler and with a much more dynamic memory management system.

Despite a considerable amount of buggy features in the language and some compatibility issues between the different browsers, JavaScript quickly became a popular language for the Web. Developers could easily create interactive behavior like changing color on a button when a mouse hovers over it, give the user feedback if the input in a textfield is wrong, etc. Much of its success was because of its simplicity; there was a low barrier to add JavaScript behavior into Web sites. Because there is no compilation

process, and no "start function", only independent functions that can easily be written and "tossed" around in the document, unprofessional developers could quickly implement exciting and dynamic features [20]. On the other hand, for this reason, many professional developers would consider the language of being strictly for amateurs and not suitable for professional developers [31]. Its strength was clearly for small sized applications, as browsers at the time were not able to execute large-scale JavaScript code. Also, considering that the JavaScript language at the time was very simple and limited, the development of large-scale JavaScript Web applications was simply not feasible.

In an effort to improve the language features of JavaScript, and its browser incompatibilities, a standardization process of JavaScript was given to the European Computer Manufacturers Association (ECMA) in 1996. The language was actually renamed to ECMAScript, although most people still refer to it as JavaScript [22]. Unfortunately, as the standardization was being developed, the browser inconsistencies, especially between Netscape and Internet Explorer continued to grow. Many JavaScript frameworks were built as workarounds to the inconsistencies, but most solutions weren't good enough. This led to alternative solutions for highly interactive and graphical client-side behavior such as Adobe Flash[4] or Microsoft's Silverlight[79].

An important part of JavaScript's history was with the rise of **AJAX** (Asynchronous JavaScript and XML) technologies[48], which gained much attention right around a century after JavaScript was first introduced. AJAX is an API that offers JavaScript functions that makes the browser asynchronously fetch data from the server without having to refresh the current Web page. Instead, the requested data would be used to alter the page. This would result in a much more interactive user experience because the browser would neither block while waiting for the result, or re-render and paint the whole DOM tree upon successful complete. Since the introduction to AJAX, JavaScript has increasingly become a very popular language, and it has also brought the attention of professional developers. This has led to successful framework solutions like JQuery[68] and Prototype[103], which simplifies the development of complex dynamic behavior and fixes browser incompatibilities, so that the Web developer doesn't have to write specialized code for each browser.

The increasing popularity of client-side application development with JavaScript has led to powerful JavaScript engines in modern browsers, making memory management and JavaScript interpretation highly effective. In September 2008, Google built the Chrome browser with its V8 JavaScript engine[126], stating that low performance JavaScript implementations are no longer sufficient. Other browser vendors followed along, and today JavaScript performance is more superior than ever. Still, however, there are drawbacks with the language itself. Even though libraries like JQuery and Prototype simplifies the development of interactive and cross-platform Web pages, it is easy to end up with a big pile of tangled JavaScript event handlers and unstructured functions (often called spaghetti code[114]). Most of the reason is that the language lacks features like classes, modules

and namespaces, which makes it difficult to develop flexible and maintainable large-scale JavaScript applications. However, a lot of work has been done lately to implement quality frameworks that provide comprehensible syntactic sugaring for the language. These frameworks use some of the nice, and for many unknown concepts of the JavaScript language such as inheritance and closures to enable a highly flexible and structured development environment. All this has led to the possibilities of building large-scale JavaScript Web applications that runs primarily in the browser. Good examples are Google's Gmail[54], and Maps[55].

Node.js [90] is a Web application framework built for using JavaScript as the programming language. This is one of the first (and most popular) solutions for developing JavaScript Web applications on the server. It runs on Google's V8 JavaScript engine. A big advantage this framework has compared to other frameworks such as Rails or Spring, is that it is single-threaded and event-driven. This is a completely asynchronous programming environment that is centered around events, where clients subscribe to events and are notified when the events are triggered. This avoids the blocking scenario that might occur in regular synchronous systems.

2.3.4 Client-server Interaction Schemes

There are multiple ways the browser can communicate with the Web server. In this thesis, we mainly adhere to two different ways. Synchronous client-initiated requests, and asynchronous client-initiated requests. These communication schemes all use HTTP, and are client-pull based. However there are other push-based alternatives such as Comet[30], and WebSocket[130], where the client and server maintains an open connection, and the server can notify the client of changes.

Synchronous In a synchronous HTTP requests, the client-browser asks the server for data, in which the browser will wait for the server to respond. The request is normally triggered by the user clicking an anchor tag (hyperlink), or submits an HTML form. This would usually result in a new HTML page that is returned to the browser, in which the browser would start a complete rendering process to build a new DOM tree, and paint it on the screen. A normal term for this is a page *refresh*, or *reload*.

Asynchronous In asynchronous HTTP requests, the client sends a request to the server, without the browser having to block while waiting for the result. Usually this happens with an AJAX request. When the result is received from the server, a browser event is triggered that is normally picked up by a JavaScript handler function. Typically the JavaScript handler alters the DOM tree with the new data received from the server.

One potential drawback with AJAX is that not all browsers support JavaScript. Examples are some smartphone devices or PDA devices. Also,

pages that are generated using AJAX are not automatically picked up by Web crawlers, because most Web crawlers do not access JavaScript code. This means that content generated by AJAX would normally not show up in public Web searches. However, in 2009, Google proposed a programming technique to make AJAX pages crawlable[101]. This is a somewhat complex technique, and can be tedious to implement in larger JavaScript applications.

2.3.5 HTTP Sessions

An HTTP-session is a semi-permanent communication dialogue that exists for two communicating entities (here, the client and the server). A session normally has a time-out value, such that when the time runs out, the session ends. The HTTP protocol is stateless in its nature, because every HTTP request is self-contained, and independent of every other request. Therefore, to be able to maintain state in an application, the client and server can incorporate a session protocol. The state information itself is data that has to be maintained between multiple pages in the application. Examples are shopping cart information in an e-commerce site, flight booking details, or authentication credentials. Imagine the user having to identify himself for each request that is sent to the server. This can be avoided if state information is persisted and being referenced in each request.

There are a couple of ways to implement sessions:

- An object that is kept on the server . This object can be referenced in the client's cookie, or in the URL, if cookies are not supported.
- In the messages sent between the client and server, for example by populating the cookies, or keeping the data in the HTTP request and response body, or in the URLs. Clearly the size of the session data is very limited in this case.
- In the browser's own storage system, thus maintaining sessions only on the client.

2.3.6 Representational State Transfer

Modern Web applications often follow a design pattern named Representational State Transfer, or simply REST [44]. This pattern states that all HTTP URL's must reference a particular resource on the backend, by using one of the HTTP request methods Get, Post, Put, or Delete (HTTP also supports additional request methods, but these are the most commonly used). This way, every URL offered by the Web application are self-contained in that it contains all the necessary information needed to satisfy a request. Resources are uniquely identified by a URI, and manipulated through the HTTP method interface. Applications that follow this pattern are named RESTful applications. A common use case for RESTful applications is to offer the self-contained URL's publicly as an API to clients other than just the

application's own front-end, like other third party applications that wishes to use the applications REST services. Further, the REST pattern states that each REST request is stateless, hence adhering to the nature of HTTP which is stateless. That means, REST requests should not depend on an ongoing session in order to generate proper results.

2.3.7 JSON

JSON[70] (JavaScript Object Notation) is a text based data format that is based on a subset of the JavaScript programming language. It is easy for both humans to read, and machines to parse, and has a similar syntax to many of the programming languages based on the C family of languages. The data structure fits well as a transmission format in Web applications, because it is both simple and light weight, easy to modify and is supported by many programming languages. The format itself is very simple, and the datatypes offered are limited to numbers, strings, arrays, booleans, and objects (being key-value pairs of the types just defined). An example of a JSON object representing a guitarist is showed below:

```
{
  "username": "Paul ShredKing",
  "age": 21,
  "country": "Norway",
  "guitars":
  [
    "Gibson Les Paul",
    "Fender Stratocaster"
  ]
}
```

The closest alternative to JSON is XML[38], which is another transmission format often used on the Web, and especially with REST communication. However, its syntax is a bit more verbose, and requires more processing to manage because of its complex markup tags and syntax rules. On the other hand XML lets one add more restrictions to the data then with JSON.

2.3.8 Business Logic and View Logic

In Web applications, one often separates two very different programming concerns; the business logic and the view logic. The business logic (also called domain logic) typically represents:

1. The application's **domain**, often called business objects. They describe the application's core entities. Classical examples are Account, User, Purchase, Loan etc
2. The operations that can be performed on the business objects
3. Interactions between the business objects, and business rules that state the values that business objects are allowed to have

The view logic (often called presentation logic) describes how the domain is visualized in the user interface. The view logic implements dynamic user

interface behavior. It is often generated on the server, and depends on the application's current state. Typical Web 2.0 applications contain complex view logic.

Separation of business logic and view logic is considered best practice[112] in order to let one concern change independently of the other, and to enhance a coherent codebase where separate concerns does not directly depend on each other[78].

2.3.9 HTML Template Rendering

In dynamic Web pages, when an HTTP request comes in for a particular HTML page, the server has to prepare the HTML page with proper content based on the data received in the request. One way to generate dynamic HTML (that is, perform view logic) is to generate the HTML directly in code as Strings, and send the result back to the client. However this approach is messy, difficult to maintain, and the developer has to know the programming language that is creating the HTML strings. In other words, not a preferable solution for Web designers who only knows HTML. The preferable approach is a process called HTML template rendering. With HTML template rendering, a template system is organized as a set of **template files** (often called **views**), some domain and state data, and a rendering engine. The template files are implemented in a special template language. This is basically just HTML with additional syntax that refers to and can operate on data variables in the Web application. The operations supported are usually limited to simple loops and conditional expressions, just enough to facilitate the injection of dynamic data without confusing front-end designers. To generate a page, the rendering engine takes as input one or more template files, the data needed to populate the templates, and produces as output an HTML page. The necessary data is often fetched from the database or exists already in the server's memory. The resulting HTML page is sent back to the client.

Recent JavaScript technologies have also enabled HTML rendering to happen in the client. The process is very similar. HTML template files can be sent to the client, which are ignored by the browser so the browser won't automatically paint them on the screen. Special JavaScript rendering engines that does the same job as the server-side rendering engine recently described can be accessed by JavaScript code in the browser. When some HTML template is to be rendered, the JavaScript rendering engine is called with an HTML template and a data object as input, and it returns an HTML page populated with the data content. The resulting HTML is typically appended to the DOM, or swapped with existing DOM elements.

2.3.10 Databases

Databases, and especially relational databases have since the beginning of Web application history been the most popular form of storing persistent data[44]. Other alternatives have also been used such as flat-file storage (where the content is stored as plain text or binary data) or XML- or object

databases. Much of the reason for the success of relational databases however, is that it provides **durability**, which in the context of data persistency means that once the data is stored, it is guaranteed to exist even if machines holding the data crashes. Also, a reason for the relational database's popularity is that it stores information in a structured format, which often fits the structured data formats that are manipulated by the Web applications. However, other types of databases that differs from the traditional SQL format has recently entered the market. These are commonly referred to as NoSQL databases.

ACID

ACID is a popular term in the context of databases. It is a set of properties that guarantees reliability when it comes to transaction management. Database management systems often state that ACID guarantees are provided in their system, in order to promise a reliable database solution. Each property is defined below:

Atomicity guarantees that either all the commands in a transaction completes, or non do.

Consistency guarantees that all the data will always be in a consistent state according to pre-defined rules. A transaction brings the system to a new consistent state.

Isolation guarantees that parallel transaction executions are always processed as if they happen serially, i.e no interference of any two parallel transactions.

Durability guarantees that committed transactions are safe, and lasts even during system errors or crashes.

ACID guarantees are often provided by relational database management systems (RDBMS). However, NoSQL databases tend to have more relaxed relations to the principles, in favor for speed and simple replication abilities.

CRUD operations

In Web application terminology, one often use the word CRUD to refer to the four essential database operations; *Create*, *Read*, *Update* and *Delete*. These are the main operations performed by the Web application, on the data that needs to be persisted. When a CRUD operation is executed, it is the application's responsibility to convert the data into a format that fits the database's technology, and vice-versa. This process is called **marshalling**, or serializing. One example is when the application is to save a new object in the database. In this case, a *Create* operation will be performed where the application will transform the object from whatever programming

language syntax the object is currently described in, into a structure that fits the given database's syntax. The application will send this transformed (marshalled) object to the database, which is now able to parse the object and save it to its storage structure.

Relational databases

Relational database management systems is a storage system based on a formalism known as the relational model. The formalism is based on structure and relationships, where the data entities are stored into **tables** that contain a set of **attributes** that describe the table. The tables can be related to each other to form groupings. RDBMS's stores a collection of tables, where each data entity is represented as a **row** in a specific table, and each column in a row represents an attribute for that entity. The most popular form of manipulating data in a RDBMS is SQL (Structured Query Language)[27]. This is a query language used to insert and manipulate data in a relational database. There are popular dialects of the language, generated by database vendors such as Oracle's SQL[93], Microsoft's MS SQL[80], MySQL[88] and the open source PostgreSQL[99].

NoSQL

NoSQL is a broad class of various database management systems who all have in common that they don't share the relational structure from normal SQL databases. The reason for its existence starts with the rise of Web 2.0 applications, when developers saw the need for simplifying replication of data, higher availability, and a new way to manipulate data that can avoid the need to perform tedious mappings between SQL strings and objects in any given programming language[71]. The main potential for NoSQL databases is to perform operations on massive amounts of data that is not structured or connected in complex relationships. Very often this applies to Web 2.0 applications, because much of the information in such applications can be gathered in coherent entities, thus avoiding the need for complex relationships and tedious operations to join them together. A typical example is users that has arrays of blog posts, and blog posts has arrays of comments, in which case all these fields are nested inside the user abstraction.

What's typical with NoSQL is that they are often customized to solve a particular type of application's persistency need. Therefore, there are many different classifications of NoSQL databases, which vary in the way they structure the data. An overview of the some commonly used NoSQL categories is summaries in the following list:

Key/value store

Is a simple database store where data is identified by a key, and the data itself can be any datatypes usually supported by the implementing programming language. The structure is schema-less, meaning it doesn't provide complex structures with foreign key constraints. It is also highly efficient as the database is often

implemented as a HashMap. One popular example is Redis[106]. This is an extremely fast key-value store that favors speed over durability. It also provides simple replication support, making it easy to distribute the database over multiple machines. Much of the reason for Redis' extremely high speed is because the data is typically being kept in memory, and only written to the database as a snapshot every once in a while. Other popular key/value stores are Riak[108] which favors scalability and fault tolerance, and Voldermort[102], which favors simple distribution (data is automatically replicated).

Document-oriented databases

Is a datastore that is based on documents that contain unstructured content. Documents are often separated into unstructured collections (can be viewed upon as SQL-tables), where unstructured here means that content in the same collection can have different structure. However there is some variation in the way the different database implementations choose to define the formats of the documents, but it can be assumed that each document encapsulates some logically associated data in a predefined format. An interesting property with these databases is that performance is often not the main goal, but rather programming satisfaction. As many of these are implemented in JavaScript and offers querying semantics and data structures based on JavaScript objects, it is really easy and flexible to perform database operations on them. Examples include CouchDB[8] and MongoDB[82].

Column-oriented databases

Is a database system where data is organized as columns, as opposed to row-oriented databases such as SQL based databases. In this scenario, every value that would usually be in a row gets its own instance in a column together with its belonging identifier (Id). As such, it is very efficient to perform range queries over a big amount of column data. Examples are Cassandra[18] and Google Big Table [21] (although these are not pure column-oriented, but rather a hybrid).

2.4 Summary

We started this chapter by looking at how the World-Wide-Web began in the late 80's. In the beginning, Web sites were primarily a collection of static pages, but gradually turned into more dynamic pages where the content changes depending on attributes provided by the client. After this we saw that some of the technologies that are found on the Web today are outcomes of the browser wars that started in the mid-90's. This especially concerns the JavaScript programming language, which has resulted in many browser incompatibilities and a somewhat misunderstanding and unawareness of JavaScript's core language features and capabilities. However, major browser vendors, and Google especially, has acknowledged the advantage of having a unified language for the Web,

something that has brought a lot of attention and improvements to the JavaScript programming language and its related technologies. This has now made JavaScript become a highly popular language for the Web, and developers have started building large-scale dynamic Web applications purely in JavaScript.

In the second part of this chapter, we looked into popular technologies that are commonly found in modern Web applications. Examples included AJAX, which offers an asynchronous client-server communication scheme, REST, which is a design pattern that states that Web resources should be manipulated exclusively through methods specified in the HTTP protocol (get, put, post, or delete), and NoSQL databases, which are databases that doesn't abide to the relational model, but instead specializes in speed and scalability through replication. We also looked at a specific type of modern Web applications, namely Web 2.0, which we saw were interactive Web apps with responsive and rich user interfaces, and social networking features.

Chapter 3

Design Alternatives for Modern Web Applications

3.1 Introduction

So far we have discussed the behavioral trends in modern Web applications and the technologies that runs them. We saw that modern Web applications are often very interactive with rich user interfaces, that looks and performs like native desktop applications with graphical user interfaces. In this chapter we look at popular ways these applications are built. This concerns the application's **architecture**, which is a concept that describes the application's structure, at different granularities.

The traditional Web application has in the last decade followed a thin-client approach where all the logic happens on the back-end. In such architectures, HTML pages are dynamically generated on the server and handed to the browser every time the client issues an HTTP request. Lately however, there has been an increasing interest in moving much of the application's logic to the client, and abandoning the server-side page generation in favor for client-side page generation. This is called a thick-client architecture.¹ Still however, a lot of applications follow the traditional approach, as many developers and application owners are skeptical to the thick-client model; this architecture implies heavy use of JavaScript, which has since its beginning had a lot of opposition, and people are still using old browsers that does not fully support JavaScript[120].

The main purpose of this chapter is to outline the differences between, and popular ways to build traditional thin-client architectures, and innovative thick-client architectures. This also includes popular database solutions for persisting data in Web-apps. The first part of this chapter is

¹The thick-client architecture is not a new idea; thick-client architectures have been around for many years, where programs are sent from the server and executed on the client. Often this would be online games, calculators, or other user-interactive applications. However, these applications depend on a specific program that is compiled and executed on the client, in other words, not traditional Web pages that uses common browser-supported technologies.

divided into two main sections, one for each architecture. After this we will see how these concepts have been used in the prototypes built for this thesis. We will refer to the traditional approach as **Reference-model 1.0**, while the latter approach will be referred to as **Reference-model 2.0**. Thus, these two will be our reference models for software architectures that are aimed to build modern, interactive and scalable Web apps.

3.2 Reference-model 1.0

Going back approximately 15 years, dynamic Web applications were often built with Common Gateway Interface (CGI) technologies[29]. With CGI, a Web server accepts URLs that are delegated to an appropriate back-end program. A process is started on the server, and the CGI program executes the given request, which results in an HTML page that is sent back to the client. This solution however, was not very scalable considering each request would trigger a new process on the server. Gradually, as the Web got more users and the applications became more complex, new Web framework technologies came along. Examples are PHP[96], Java EE[64], Ruby on Rails[109] and Microsoft's .NET[59]. For many years, developers have been building Web applications with these technologies, where all of the application's logic is executed on the server. This implies that the back-end implementation has many responsibilities, and the front-end is simply a thin-client that doesn't need to do much processing. This is only logical, as back-end implementations run on powerful Web servers, and client devices has up until recent years not been able to perform demanding processing jobs.

3.2.1 The Three-Layered Architecture

A classical way of separating concerns in a Web application is to divide the whole system into three different software layers. There are some variations to how these layers are separated, but in this thesis, when we refer to a three-layered architecture, we follow the layering structure outlined by Brown et al[6]. This architecture separates the system into a presentation layer, domain logic layer, and a data source layer. The layers reside exclusively on the back-end of the application. The front-end has little responsibility in this architecture, as its only task is to display the result that is produced on the back-end. The layers are designed to be very loose-coupled. This is done by avoiding that a module in one layer depends on a concrete implementation in a lower layer. Instead, they depend on abstractions (interfaces) which can easily be swapped out. This principle is often referred to as the dependency inversion principle [42]. The abstractions are not hard-coded into the layers, but are "injected" through function arguments so that the same function can be called again if one wants to change the type of a dependency. The benefits from having individual implementation details encapsulated in different layers, is that it is very easy to modify one layer without harming

another, and components can easily be reused. This facilitates a flexible and maintainable codebase[23].

The Presentation Layer is the application's main entry point. Each URL offered by the application is mapped to a dedicated handler (often called a **controller**) in the presentation layer. In modern Web apps, the presentation layer is often implemented with the **Model-View-Controller** pattern[107]. This is an architectural design pattern that organizes the structure of the layer. In this pattern, **controllers** handle HTTP requests from the client and simply delegates to a proper business operation in a lower layer. The result from the business function is returned in terms of **model** objects, which implement the **domain** of the application. When a business function returns the controller, it looks at the result to determine which **view** to return back to the end user. The view might be an HTML page, an HTML template file, or another data-format like XML or JSON. The latter two formats are most often used if the request is an AJAX request. If it is a template file, the view is sent to a rendering engine before the resulting HTML is returned to the client. Often, template files have references to other template files, in which case these will be merged together by the rendering engine. This facilitates decoupled view logic with fine-grained templates that can be reused.

Another architectural structure is the **Application controller pattern**[43, p.379-386]. This separates how objects are to be presented in the app, from the app's business operations, by adding a new layer which is responsible for deciding which page to show in which order. This structure is nice if there is a lot of logic required to decide the page's ordering and navigation scheme. However, as the most common architectural pattern for the presentation layer is with the MVC pattern, we will continue the discussion with this pattern in mind.

When a URL request comes in through the presentation layer, it might go through a number of **filters** before control is handed to the controller. Filters can have different objectives like authentication, marshalling of different Web formats into an object in the programming language that is used, error handling or HTML form validation. Also, the presentation layer would check the URL for a session identifier in the request's cookie, or the Url itself. The session identifier is referencing a session object that is already residing on the server's main memory, or in a database. The session object can also be used to store state information for the User, and to hold authentication details. After the request has passed the filters, the appropriate controller handler is called, which delegates control to a business function, typically in the domain logic layer.

The Domain Logic Layer , also called the business logic layer, or service layer, is responsible for executing the business operations that are supported in the Web application. These are the functions that makes up the core services of the application. Like in the presentation layer, the domain logic layer operates on the domain objects (i.e models). The

business operations in the domain logic layer uses the domain objects by executing operations on them. Examples are calculating the total price for an order of books, registering a new friendship between two users, or searching for recommended movies for a currently logged in user. The domain logic layer sends and receives domain objects from the data source layer in order to persist them in a database.

There are multiple ways of organizing how the business processes are implemented in the domain logic layer. A simple structure is by using the Transaction Script design pattern [43]. In this structure, each business operation is implemented in a single self-contained and independent procedure (script). Each procedure is mapped to an operation in the presentation layer, that takes input from the presentation layer, performs business logic (e.g data validation, calculations etc) and stores data in the database. This approach is very simple, as there is no domain abstraction and complex object-structure, just independent functions. However as applications get complex this pattern might lead to code duplication as different transactions might have common behavior. Another more common and object-oriented structure is to use the **Domain Model design pattern** [43] where business logic is encapsulated in domain objects. For example a BookOrder class would have functions for creating a book order given a list of books and a user id, fetching a book order given an order Id, updating a book order, and deleting a book order. In addition, the domain objects encapsulate the data attributes that represent the state of the object. The domain objects can have dependencies on each other, and call each others function in order to gain code reuse.

The data source layer is responsible for communicating with other systems, such as databases, messaging systems, external Web services, the filesystem etc. Traditional Web architectures uses a relational database management system, which is the most popular solution for choosing how to persist the data in modern Web-apps[89]. Thus our Reference-model 1.0 will define that data is persisted in a relational database. The reason for its popularity is much due to the relational model, which brings a flexible and highly efficient query language (SQL). Most computer science courses on databases teach SQL, so developers tend to think data relationally. Also, data safety guarantees are provided with ACID, and the wide offerings of relational database management systems with its many development toolkits makes it a natural choice. Not to mention, the technology itself is many decades old and therefore brings years of experience and documented best-practices.

The data source layer is responsible for marshalling domain objects into proper storage representation (also called database mapping), and vice-versa. For instance translating a SQL table into a Java object. The data source layer has to connect to the database, handle database transactions and close database connections. Usually this is handled by the Web application framework, so the data source layer only has to worry about how to perform operations on the database. As with the domain layer,

there are two popular design patterns for structuring the data source layer. One is with the active record pattern [43], in which case each domain object would know how to perform CRUD operations on themselves. Another approach is the data mapper pattern [43] where there is one separate class for each domain object, that performs marshalling of the given domain object, and is responsible for implementing CRUD operations on behalf of its domain object. The data mapper pattern separates the persistence code out of the domain objects, but adds more classes to the system. The active record pattern has a tendency to grow big in size, if the domain objects has to support a large amount of complex database operations.

There are also frameworks that perform the marshalling, given some simple configuration of the domain objects. These are called object-relational mapping (ORM) tools. They often provide caching mechanisms to avoid using the database as much as possible, and they allow the programmer not to worry about marshalling at all. In many cases this could lead to less code in the data source layer. [71] Examples are Hibernate [58] for java, MyBatis[87] for Microsoft .NET and Java, and LINQ [72] for Microsoft's .NET. It is important to point out that even though these frameworks hide the complexity of object-relational mapping, it does have some pitfalls. Many developers argue that by using ORM-tools you loose the ability to exploit the full features of a database management system[115, p. 124]. This includes the ability to do customized database tuning, and take advantage of special data types that are supported by specific vendors. Plus, the fact that an ORM-tool does indeed hide the object-relational mapping code, makes it harder to debug, and also, there might be some performance overhead due to the complex code that is generated by the ORM framework.

3.2.2 The Front-End

Up until now, we have been discussing the back-end implementation of a traditional Web app. The front-end consists of the set of HTML pages, CSS style sheets and JavaScript files that makes up the user interface of the application. User-navigation in traditional Web apps is often done through HTML forms or hyperlinks, which leads to a new page that is completely rendered on the browser (a page refresh).

Also, in cases where a highly interactive event is to happen, a dedicated JavaScript event handler that is registered to listen to certain events will perform the action. This could be displaying a pop-up window or an animation effect on a mouse-hover event. The event handlers are registered with the browser when the HTML page is first rendered. In traditional Web apps, these handlers are often self-contained and independent JavaScript functions that has no structure or modularity. In most cases, this is because of some developer's relaxed relationship to the language; the JavaScript code is developed by adding function after function that merely serves to implement a new dynamic feature. This often leads to spaghetti code[114], in cases for applications that include a lot of interactive JavaScript behavior[74].

3.2.3 Platform Environment

Another aspect of the classical three-layered architecture is application tiering.² A Web application is often divided into three tiers; the presentation tier, the logic tier, and the persistence tier.

The Web server hosts the presentation tier which communicates with client users through HTTP. The Web server listens on port 80, which is the port number used for HTTP. The HTTP request is forwarded directly to the appropriate code on an application server. The application that runs on the application server communicates with the persistence tier that is usually hosted on one or more database servers. In a bigger production environment, it is normal to distribute the tiers into separate physical server machines (called horizontal scaling). Each server is hosted on a separate machine in a so-called **shared nothing** manner, meaning the servers on each tier are independent so they don't have to communicate with each other. This makes it easy to add more servers on demand without any synchronization difficulties. Note that when the persistence tier is composed of a relational database management system, a shared-nothing architecture is difficult to implement, due to the nature of the relational model [2]. This makes it difficult to do horizontal scaling with relational database systems.

However, some SQL systems are particularly designed to scale vertically[19], and many large-scale Web 2.0 applications does successfully implement shared-nothing architectures backed by multiple distributed SQL databases. One example is the social-networking site Pinterest[97]. A report[111] outlined their platform environment, which among others include:

- 88 MySQL servers
- 180 Web servers
- 240 Application servers
- 200 cache servers

3.2.4 Examples of traditional Web architectures

In this section we look at two examples of some common, traditional three-layered Web application architectures. The examples use Web application frameworks that is made for different programming languages, and promotes different architectural solutions. The purpose is to propose a set of popular Web architectures, that will be used as a base to determine the architecture we use to design the first prototype in this project.

²Application layering is a term that divides the code into separate logical software layers. Application tiering on the other hand, is another logical separation often associated with where these "tiers" are physically deployed. For instance a three-tiered Web application could have its presentation tier on the Web server, the logical tier on a dedicated application server, and the persistence tier on a dedicated database server.

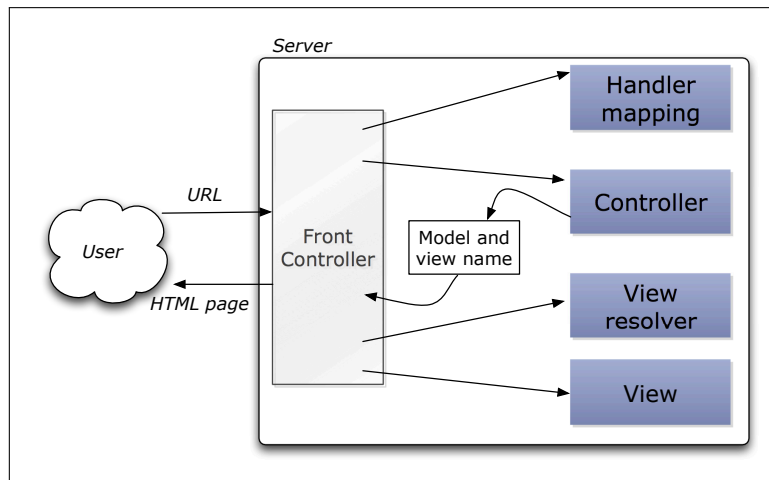


Figure 3.1: A request flow with spring MVC

MVC with Ruby on Rails . The Rails framework for Ruby has become a highly popular backend technology for Web applications, having many big commercial users. The framework is built around the MVC design pattern [107]. In a Rails application, the controllers act as thin classes that receives a URL request, and delegates business logic to the models. A model is a Ruby class that implements the Domain Model design pattern. The views consist of HTML template files, often written in a template language like HAML[57] or Mustache[86]. These templates can reference data variables in a model class, and are rendered into HTML files on the server before they are sent to the client's browser.

The data source layer in Rails is typically built with the Active Record design pattern. This works by letting the model objects implement CRUD operations for manipulating the database. This way, every model class contains business logic operations and the functions required to persist and manipulate the particular model in the database.

Front Controller with SpringMVC SpringMVC is a Web framework that wraps the Java Servlet API technology. The Java Servlet API is a set of classes that implements low-level protocols to communicate on the Web (e.g HTTP). For example, the Java Servlet API implements an interface called *javax.servlet.http.HttpSession*, which provides session management. This session implementation puts an Id (called the JSESSIONID) in the User's HTTP cookie. This Id refers a particular HttpSession object that belongs to the User, and it is maintained by Spring on the server.

Just like with Rails, SpringMVC is also built around the MVC pattern. However, the framework also applies an architectural pattern called the Front Controller pattern[41]. This pattern works by having one central servlet (the front controller) that receives all HTTP requests, and delegates control to a set of components that handle the request. This can be seen in figure 3.1.

When a request comes in, the front controller will ask a **handler mapper** to get a reference to a specific controller based on information provided in the request URL. A controller is a SpringMVC component (related to the controller in the Model-View-Controller pattern) that is responsible for processing the actual requests. The handler mapper, is another Spring component that in addition to knowing what controller to issue based on a URL, performs pre- and post processing (i.e filtering) procedures, such as HTML form validation. When the handler mapper is finished pre-processing a URL request, it returns a reference to the right controller. The front controller then delegates the request to this controller. The controller typically just delegates to a business operation, and upon completion, it populates a model object with necessary data that is to be displayed in the view. The model object is a simple key-value data structure that lets the developer easily reference its data from the view file. The controller function also returns the name for the view that is to be rendered together with the model object. The view is a template file, often written in a template language such as JSP[67] or Velocity[118]. It is the view resolver that maps a logical view name (e.g "homePageView") to a physical view name (e.g "/WEB-INF/views/homePageView.JSP"). Finally the view and the model object is rendered to HTML and send back to the client.

With Spring comes a great implementation of an inversion of control container (IOC)[42]. Inversion Of Control is a programming methodology where the concrete types of object references are not known at compile time, because the references are instantiated and populated by an assembler at run time. This avoids having tight couplings between classes, and promotes a flexible codebase. The IOC container is a module that is responsible for creating objects, populate their references to other objects, and manage their complete lifecycle. The container is fully configurable, which makes it easy to decide how the classes are instantiated. For example, objects can be configured to be lazily instantiated, meaning the object won't be instantiated until it is first referenced. Also, there are multiple ways Spring's IOC container can be configured to create objects. In Spring, this is referred to as the object's **scope**. Some common scope-alternative are:

- Request - one instance is created for each HTTP request
- Session - one instance is created for each HTTP session
- Singleton - only one instance of the given class is ever created.

A typical domain logic layer in a Spring application is built with the service layer pattern [116]. In the service layer pattern, the business logic is split into two. A service layer that exposes an API that encapsulates all the business operations, and the domain model which encapsulates the application's domain in separate classes that merely keeps data variables. The API, or service layer is categorized into logical abstractions called **services**, where each abstraction is hidden behind a facade [47, p. 158]. A facade is an interface that contains simple access methods to a more complex set of data structures, like complex business operations

or database access methods. Hence, each service encapsulates complex business operations and communicates with lower layer data source functions. The domain objects typically has no logical functions, only private data attributes with accessor methods, as opposed to the domain model design pattern. The service classes are also responsible for handling transaction management, so that if a transaction fails, the service classes know how to handle it. When performing database transactions, the service classes delegates to the data source layer which would know how to communicate with the database and perform object-relational mapping. This could be done by implementing a customized object-relation mapping scheme, for instance by implementing the data mapper pattern described earlier, or by using an ORM tool such as Hibernate or JPA[121]. The latter case is similar to the active record pattern, often used in Ruby on Rails. Spring is one of the most popular frameworks for Java, as of 29th April 2013[59].

3.3 Reference-model 2.0

The motivation for proposing a reference model for modern Web app architectures has its roots in an architectural shift that started around 2010, when browser's capabilities to execute JavaScript increased tremendously. This was by the time when Google launched its Chrome browser with the powerful V8 JavaScript engine, and the compelling browsers followed along with similar JavaScript capabilities. Also, The JavaScript language itself has started to get much more endorsement from the Web community with the standardization of ECMAScript, and Google's provenly working large-scale JavaScript applications like Gmail, Google Maps and also Node.js. This has led to many new experimental Web architectures that takes a distance from the thin-client model, and where application logic is gradually moving from the server and into the client. This means that the back-end is left being a simple and agnostic storage center for persisting the domain data in the application. Not only is this feasibly, but it's becoming increasingly popular. However, many developers have a skeptical relationship to JavaScript, partly because JavaScripts history of buggy features and browser incompatibility problems. Also, many developers are not aware of JavaScript's features like object-orientation including prototypal inheritance, and functional- and dynamic programming facilities, having closures and a dynamic typing system. Instead, they have acknowledged the fact that building large-scale JavaScript applications doesn't scale in terms of code maintainability; it often ended up as piles of spaghetti code[74]. This is an unfortunate misinterpretation.

Together with this thick-client approach one has also seen a sudden interest in alternatives to the traditional relational database. With the increasing popularity of applications being deployed and run in the cloud, there is a need to be able to distribute an application's database over many servers. Now, because traditional SQL databases has showed not to replicate very easily[2], this issue, together with a need for a

simpler programming interface against the database, has led to the many alternative NoSQL databases.

In this section we propose an architectural approach where the application is moved to the client using JavaScript, and where data is persisted with NoSQL technologies.

3.3.1 Front-end Frameworks

An interesting aspect of modern Web application development is the evolution of JavaScript development environments. Not only has JavaScript been judged for being a language with many limitations, but it has also lacked proper frameworks and plugins for simplifying development of large-scale applications. However with the increasing interest for such JavaScript applications, a huge amount of frameworks and language variations have been built. This includes:

- Frameworks for structuring and organizing JavaScript code.³
- Programming languages that compile to JavaScript, to facilitate the development of large-scale JavaScript applications with a language that is more similar to traditional languages like Java or Ruby. Examples are Coffescript[28] and Clojure script[26].
- Frameworks for syntactic sugaring of the JavaScript language, useful mathematical operations, fixing browser compatibility issues, and simplifying working with the AJAX technology [113]. Popular examples are JQuery[68], Dojo[35] and Backbone.js[14].
- Rendering engines for the front-end, built to produce HTML given a template file and data objects.

3.3.2 Thick-Client Concepts

Having the application moved to the client means that the one-to-one mappings between user interactions and controller handlers on the server are gone. Instead, these events are now picked up by JavaScript handlers on the front-end. The front-end has taken over many of the concerns that used to be implemented on the server. This includes:

- Routing between pages
- Render views into HTML
- Sessions and state handling
- Business logic operations
- Deciding what to store in the database and when

³There is an project[124] on www.Github.com[50] where developers are implementing the same Web application with different JavaScript frameworks to help developers choose a proper code organization framework for their Web apps.

Now, there might be other concerns that can be moved to the client as well, such as language translation of content and third-party API requests. However, I decided that this goes out of scope for this thesis. Also, there are variations in to what extent all of these responsibilities are performed on the client. For example, developers at Airbnb[5] found that they could benefit from letting the server be involved in routing between pages [16]. Also, Twitter [125] found that letting the server participate in page rendering had good performance benefits[129]. However, the main motivations for moving code to the client is because one might achieve:

- Better response-times, because tedious server requests can be avoided
- Better scalability, because less work has to be done on the server

Moving the Application to the Browser

In Reference-model 2.0, the application relies on the front-end, and is completely written in JavaScript, or a language that is compiled to JavaScript such as CoffeeScript, or ClojureScript. The JavaScript code can either be sent to the client all at once when the Web application is first accessed, or parts can be lazily fetched when needed. This requires the source code to be split into separate code files so they can be sent individually from the server. This might be a performance benefit in case the whole JavaScript codebase is very big. A pitfall however might be that very many small JavaScript files would be required and sent simultaneously, in effect potentially causing tedious transmission times. In some cases the TCP connection overhead might be a performance bottleneck because the browser usually creates a new TCP connection every time the browser requests something from the server.

Single-page Web Application Architecture

One essential advantage with the thick-client architecture is that server requests might be limited. In the traditional approach, each user interaction with the page leads to a server request that result in a new page, and the browser has to reload the whole page. This causes a disruption in the user experience. With the modern approach, the request goes straight to JavaScript event handlers. This way, the client stays on the same page during the whole session, requiring no new page reloads. If for instance a link in the navigation bar that leads to a different page in the application is requested, everything is done in the browser by manipulating the DOM tree so that the new page is displayed. This could lead to a much more fluid user experience, because server requests can be avoided, and the browser does not have to reload the entire page. This principle is commonly referred to as a Single-page app[110].

If the front-end needs to synchronize data with the database it will send an asynchronous data request to the server with AJAX. This could for instance be to save some data, or get some new data that needs to

be displayed on the page. The client can also store data in the browser's memory, such that the more domain objects stored in the browser's JavaScript memory heap, the less requests has to be sent to the server. Depending on the application, write, update and delete operations will always sooner or later have to lead to a server request, so that every user has an up-to-date view of the data. In applications that require all updated data to be available as close to real-time as possible, the data has to be directly written to the database, in effect work as a write-through cache. In applications where this requirement is more relaxed, the front-end can choose to perform persistency at a later and more appropriate time. For Web 2.0 applications, the former is often wanted, because users usually want to see the latest updated data at all times.

3.3.3 The Simplified Back-end

The responsibility of the backend is mainly to manage the database. It's interface is still exposed as controller handlers, however these are not customized for particular HTML form requests or hyperlinks that leads to a new HTML page. Instead, the backend exposes an **API** for manipulating with the application's domain in the database. This API contains a set of public functions where each function is identified by a specific URL. Each URL refers to a domain entity in the application, and an operation that the server is to perform on the domain object. Note that this operation is usually not a complex business operation, but merely a single database operation. The operations offered by the backend API are commonly expressed using merely HTTP methods. Hence, the server API is a RESTfull service that adheres to the principles in the REST design pattern. Now, instead of creating and returning a complete HTML page upon each client request, the server would return a more fine-grained data object represented in a uniform data format such as XML or JSON. This is a much more general-purpose solution, because external clients like mobile applications and other third party applications can now use the service offered by the application, and so choose how to use and display the returned data.⁴

In respect to Reference-model 2.0, the back-end can be implemented using basically every server-side Web application framework, considering the responsibility of the back-end is so very simple. Popular choices are among others Ruby on Rails, SpringMVC and Node.js[59].

REST API's and JSON

The thick-client model avoids letting the user communicate synchronously with the server. Instead, the JavaScript application that runs in the browser is responsible for knowing when it needs to communicate with the server. This would be whenever some domain objects that are not already in the browser's heap are requested, or some domain object must be persisted

⁴A typical service-oriented architecture

Method	URL	Description
Get	www.shredhub.com/shredder/1234	Get shredder with id 1234
Post	www.shredhub.com/shredder/?name=Jude Swayer	Add shredder with name Jude Swayer
Put	www.shredhub.com/shredder/1234?country=Sweden	Update shredder with id = 1234 set country = Sweden
Delete	www.shredhub.com/shredder/1234	Delete shredder with id 1234

Table 3.1: A simple REST API

to the database. The requests to the server are exclusively done through the RESTful API. This means that all domain objects that are to be offered by the server, must be accessed through one of the HTTP methods *Get*, *Post*, *Put*, or *Delete*. An example of a RESTful API that offers functions for persisting a Shredder object is showed in table 3.1. A shredder is a guitarist in the Web app prototype that has been created in this thesis.

The server would respond with the domain objects in JSON format, instead of a complete HTML file. This way, it is up to the client how to visualize the result-data. Also, the API is very consistent, because it adheres to a common interaction scheme, namely the HTTP request methods *Get*, *Post*, *Put*, and *Delete*. This creates a familiar and easy-to-understand server API. This programming interface works really well with the thick-client model, because the client tier can be completely responsible for maintaining the application's state, and thus the server can be stateless.

Modern REST API's very often use JSON as the transmission format, because it fits well into the programming model both on the front-end and back-end, because considering that the front-end code is implemented in JavaScript, and JSON is part of the JavaScript language, it is very appropriate to use JSON as a transmission format because no marshalling has to be done on the client. This might also apply on the back-end: Some NoSQL technologies stores JSON-like objects, in which case no marshalling would be needed if the programming language used supports JSON.

3.3.4 Modular JavaScript

A modular codebase is made up of highly decoupled, encapsulated pieces of coherent features that are implemented in separate modules. A codebase that consists of loosely coupled modules, facilitates a flexible and maintainable system, because the codebase contains less dependencies[23]. This makes it easier to change one part of the system without harming any other.

The JavaScript programming language does not have module features built into the language. This means that it is up to the developers

themselves to develop some sort of module framework. Various design patterns have been proposed to establish standard ways of developing modules, like the module and sandbox pattern [95]. These are patterns that gathers related code into coherent modules, fairly similar to classes in traditional object-oriented languages. A lot of work has been done to provide open solutions for JavaScript developers to build modular JavaScript code in the browser. A common solution is Asynchronous Module Definition (AMD)[13], which is an interface proposal for how to create modules in JavaScript. Having the JavaScript code separated into modules means that these modules can be split into separate source files and have references to each other. That is what facilitates the lazy loading of JavaScript files previously mentioned. AMD makes it possible for the modules to depend on each other, and also on HTML templates, so that whenever a JavaScript module is fetched from the server, the HTML template will be fetched as well, and will be available as a text string inside the JavaScript module.

The AMD principle was also made to have a better alternative to loading scripts than the traditional group of `<script>` tags embedded in HTML files. The problem with this approach is that it doesn't say anything about the loading order, meaning if any of the scripts depend on each other, there is no guarantee they will be fetched in the right order. AMD brings an API that defines all the dependencies for the modules. As such, when a module is needed, all its dependencies are first loaded asynchronously, and when they're all received from the server (or some other external source), the dependencies are made accessible inside the module. The AMD API comes with two functions: *require()* and *define()*. *Define()* is used to encapsulate a JavaScript module, and make it globally accessible, while at the same time define the other modules it depends on. The *require()* is used to asynchronously load modules into a function, in which the function will not be called until all the modules are loaded and ready to be used inside the function.

3.3.5 Client-side Page Rendering

What is special with Reference-model 2.0 is that rendering HTML is no longer a matter of rendering a complete HTML page, but rather render the parts of the HTML page that need to change, and render this immediately without consulting the server. One approach is to write blocks of HTML with JavaScript strings, and write this to the DOM when a page needs to change. However, this is not very clean, or maintainable, especially when building large applications.

Using HTML templates are preferable as these can be reused, are easy to read and can be cached in the browser. Whenever the HTML need to change, a JavaScript rendering engine will be consulted which performs the rendering. The result is either appended to the DOM, or swapped with current DOM elements.

As previously mentioned, the AMD model makes it possible to have JavaScript modules that depend on HTML template files. This facilitates a

nice programming model, because if the HTML pages are also separated into small independent templates, then these templates can be stitched together to form complete HTML pages. As such, HTML templates can be reused, removed or swapped out from the current HTML page by the JavaScript renderer. This enables a highly flexible way of altering contents of the HTML page, and efficiently altering large parts of HTML without consulting the server.

3.3.6 Client State and Navigation Handling

Another part of Reference-model 2.0 is how the state is being kept between requests. The major goal of Reference-model 2.0 is to move much of the application logic from the server to the client. Thus, being able to keep the state client side is of high priority. In Reference-model 2.0, we propose an alternative solution to this by using HTML 5's Web Storage[61]. The HTML 5 Web storage is a standardization made by W3C that offers a way to store data in the browser between page requests. It is supported in all modern browsers. HTML 5 Web storage contains two storage containers: **localStorage** and **sessionStorage**. The difference is that local storage is being persisted even when the browser is closed, and it has no expiration date. The session storage is only kept in the browsers memory until the session is over, which means either if the user closes the tab or the browser. The Web storage enables developers to store lots of more data than what it supported with cookies. As an example, Internet Explorer 8 allows for sessionStorage up to 10 mega bytes, while a cookie is generally limited to 4 kilo bytes. The sessionStorage consists of a key-value data structure that is accessed by a simple JavaScript API.

A session implementation can be built by letting a JavaScript object be created when the Web application is first accessed by the client user. The object is populated with user data, and so each time the client tier changes state or receives some state information from the server, it can be persisted in the session object. Thus the server does not have to maintain a session object in memory for each user that is currently logged in to the Web application.

Navigation is done without consulting the server, meaning all User interactions that would normally result in a new page is now picked up by a dedicated JavaScript component. This component is often called a **router**. The router's job is to ensure a new page will be rendered in the browser (typically by delegating to a proper view-logic implementation), and change the browser's URL to match the new state.⁵

3.3.7 Alternative Front-end Design Patterns

There are many ways in which to structure the JavaScript code that runs in the browser. As we did for Reference-model 1.0, we will look at some popular architectural design patterns for structuring front-end code.

⁵Changing the browser's URL recently became much simpler with the HTML5 browser history API[60], which enables developers to change the URL with JavaScript.

- **MVC/MV***: Traditional Web-apps often implements the MVC pattern. In thick-client architectures, this pattern is often used, however with some variations. Models represent the domain data and communicates with a back-end API, while views contain logic that handles the user interface (**view logic**). In Reference-model 1.0, view logic was done both in JavaScript handlers on the front-end, and controllers and HTML templates on the back-end. Controllers handle routing between views and models. However, MVC-style controllers in server-centric applications doesn't always transfer directly to the thick client architecture. Controller behavior is often implemented in both views, and the router. Therefore, a popular way to define a thick-client's architectural pattern is simply MV*, meaning models-views-and something else that's up to the developer.
- **MVP**: Is a pattern that decouples the views from models by introducing a mediator called the presenter. In this pattern, the view's responsibilities are merely thin containing little to no logic, all of which is done by the presenter. The presenter's responsibility is to handle all presentation logic and routing, and communicate with a persistency layer on behalf of the models. The pattern is mostly used in cases with complex views and many different user interactions, such that handling logic and routing can be separated and reused as much as possible in the presenters.
- **MVVM**: Is a pattern that adds an abstraction called the view-model. Its responsibility is to turn models into their user interface representation, and delegate commands from views to models. Thus the views don't have to worry about how the models should look like, or how to delegate business logic. The pattern is also suitable when models must have many different view representations.

3.4 The Solutions Chosen

The two reference models just described are popular approaches to how developers design and implement modern, interactive Web apps. In this thesis, the goal is to compare these two approaches, in order to identify their strengths and weaknesses. Therefore, the main concepts from these reference models are applied in two different software architectures for a prototypical Web app. The first solution is called **Architecture 1.0**, while the latter is called **Architecture 2.0**.

Architecture 1.0 is a thin-client application, where the application relies on the back-end. The back-end is built as a three-layered architecture. This means that all the view logic happens in the presentation layer on the server (and some is made with JavaScript on the front-end), business logic operations happens in the domain logic layer, state is being kept on the server using HTTP sessions, and the front-end tier is tightly coupled to the server such that each HTML form or link has a corresponding handler on the server which serves to generate a new HTML page given the result

Reference-model 1.0	Reference-model 2.0
Server-side page rendering	Client-side page rendering
Application logic is on the server (thin-client)	Application logic is in the browser (thick-client)
Session state stored on server	Session state stored in browser
HTML Form-based interaction with complete HTML pages returned to the browser	RESTfull AJAX requests for JSON objects and fine-grained HTML templates used to alter the DOM
SQL database	NoSQL database

Table 3.2: Comparison of the two reference models

of the request. The back-end is built with Spring MVC, and is therefore a Java Web app. It uses a SQL database to persist data, and does not use any ORM tool to perform database mapping.

Architecture 2.0 is a thick-client architecture where the application resides on the front-end. State management, controller handling and business logic all happens in the browser, which is built purely with JavaScript. The front-end uses the MV* pattern, where models are active record objects that uses the back-end only as a simple data repository. The back-end is offering its services through a Rest API, which is built with Node.js. The Rest API manipulates the database, which is a MongoDB database.

3.5 Summary

In this chapter we have discussed two very different Web architectures. A short comparison of these two is given in table 3.2. The table sums up the major differences between the two architectures, where each row concerns similar architectural issues.

Reference-model 1.0 had a thin-client model with all the business logic performed on the server. The server's job was to perform the business operations, execute database operations and create HTML pages. This is a common solution to building Web apps. Reference-model 2.0 is a thick-client architecture where most of the logic is performed in the client's browser, primarily using the server for database manipulation. A flexible thick-client codebase can be achieved by using popular architectural patterns such as MV* and MVP, and by using syntactical sugaring front-end frameworks. The RESTfull architecture, together with asynchronous HTML/JavaScript loading might result in less data sent between the client and server.

Reference-model 1.0 uses a traditional SQL database to persist data, while Reference-model 2.0 uses NoSQL databases. The latter approach is more suited for replication, and might provide a simpler programmer interface. SQL, however, is the most popular persistency solution for Web applications. At the end of the chapter we stated that the two reference models are used as a base for designing and implementing the

two architectures that have been built for this thesis.

Part II

The Project

Chapter 4

Shredhub, a Web 2.0 Application

This part of the thesis covers the prototype that has been designed and built in order to compare the two architectural approaches. The Web app has been built twice from the ground-up, using two completely different architectures. The Web app contains common features found in traditional Web 2.0 applications. This includes:

- Social networking interactions:
 - Users have their own profile account that is visible to other users
 - Users connect to each other
 - Users can create blog posts, and rate and comment on other posts
- Interactive behavior with rich user interfaces
- Large amounts of persisted, user-generated data

The rest of this part is separated into three chapters: In this chapter we look at the Web app itself, seen from the end-user's perspective. In the last two chapters, we will have a detailed look at the two ways the app was built. For the record, during the discussion we will use the term *User* to refer to a currently logged in user.

4.1 The Concept

Shredhub, is a social Web application for musicians, aimed primarily for guitarists. The application enables users to share their skills and musical passion in a social and competing manner. Through a modern and interactive user interface, the users are able to post videos of themselves playing a short tune. Everyone can watch, comment and give a numbered rating to the videos, so that the creator can achieve experience points and become highly ranked on this social platform.

Now, it is important to acknowledge the fact that this application is primarily for guitarists, which does imply a slightly small user group. A better solution would be to implement a system that supports more kinds of musicians, for instance drummers, piano players, saxophone players etc. Therefore, a better solution could be to let the users pick their preferred instrument before they access the application's main page. From there on they would only be able to participate with the kind of musicians the user picked at startup. However, because I have only had a certain amount of time to implement this application, extending the application unfortunately goes out of project scope. Therefore I content myself with only supporting guitar players in this project.

4.2 User Functionality

There are many terms and concepts used on Shredhub. Here is a general overview:

- A **Shredder** is a user on shredhub. The shredder has a profile that includes (among other things): profile-image, list of guitars, list of equipment, home-country, experience points etc
- **Experience points** is a rating of how skilled a Shredder is. This rating changes when someone rates a Shred the Shredder has uploaded
- A **Shred** is a video of a Shredder playing a short tune. The Shred has a set of tags ¹ that categorizes the video. Other Shredders can rate and comment the Shred. Also, Shredders can remove a comment they made.
- **Shredders** connect to each other in fan-relationships, meaning a Shredder A can be a fan of Shredder B, such that Shredder B is a fanee of Shredder A.
- Two Shredders can **Battle** each other in a Shred-battle. This is a turn-based game where Shredders upload Battle-Shred videos in a specific battle-category. Others can rate the Battle-Shred videos, such that the purpose is to have a highest summed up rating.

4.3 Pages and User Stories

Given below is the set of pages and user stories in Shredhub. Each page is outlined together with its URL, and a user story that is found on the page. Notice that the URLs are not real in this discussion, they are just fictive examples.

¹Tags is a widely adopted term in the world of Web 2.0[128]; many web 2.0 applications use tags to classify things like blogs and images

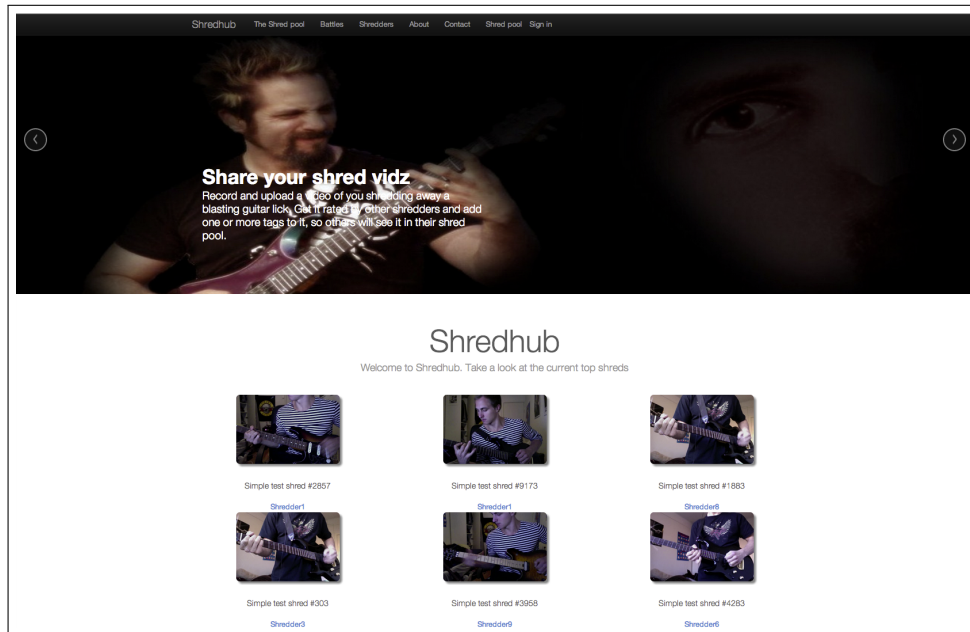


Figure 4.1: The front page at Shredhub

The front page, www.shredhub.com

The User is first met with a front page as seen in figure 4.1. Here the User can either register as a new Shredder, or log in with a username and password. The User will not be able to access any of the other services in the app before he is logged in. The page also displays a set of the current most popular Shred videos.

The shred pool, www.shredhub.com/theshredpool

This is the first page the Shredder meets when he logs in. It is the “main-page” on Shredhub, which contains multiple rows of Shreds made by other Shredders. The page contains the following rows of Shreds:

1. The latest Shreds
2. Shred-news:
 - (a) Newest Shreds made by fanees
 - (b) Newest Battle-Shreds by fanees
 - (c) Newly created Battles by fanees
 - (d) New recommended Shredders to connect to
3. Shreds with particular high rating
4. Shreds from Shredders that might be of interest
5. Shreds based on tags the User enters

Every row contains a collection of 3-5 Shreds, except the final row which contains 20 Shreds. The User can click the next button in a row,

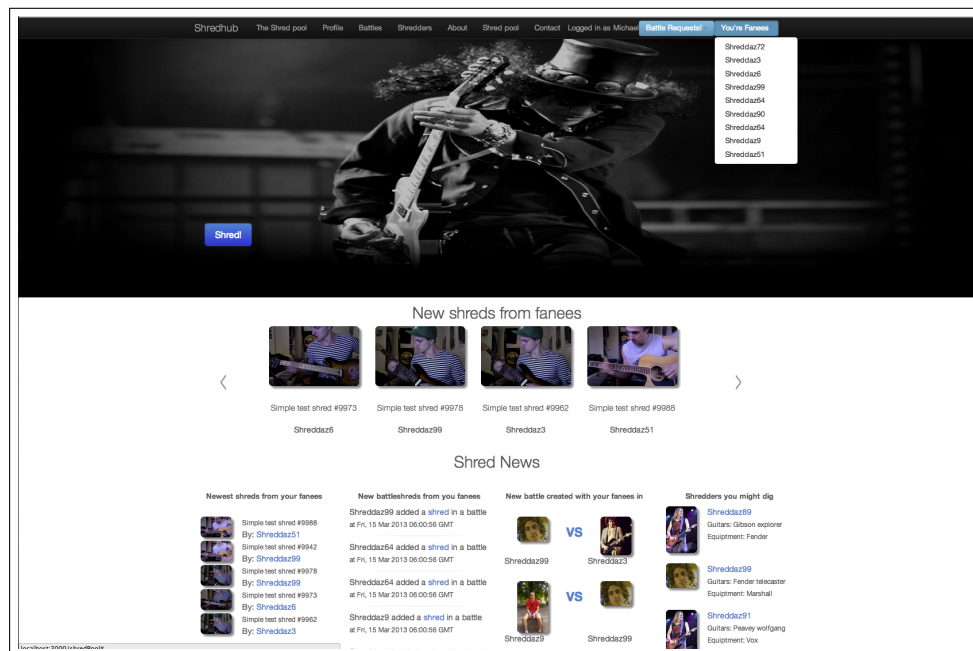


Figure 4.2: The Shredpool (top)

which results in a new row of Shreds. The Shred-news section is a set of Shreds and Shredders especially picked out to fit the User's profile, that is, new content made by his fanees, and recommendations for new Shredders. The Shredpool is showed in figures 4.2 and 4.3 on the next page

The shredder can also create and upload a new Shred by clicking "Upload shred". If the Shredder clicks on a particular Shred in any of the rows, a new window pops up displaying the Shred video.

Shredders, www.shredhub.com/shredders

This is an overview of all the the Shredders that are using the app. Considering that the amount of Shredders on the page might be very big, the list is paginated, meaning a fixed number (20 in this case) is displayed at a time, and the Shredder can click next to iterate to the next page of Shredders. Shredders can also search for other Shredders by name. The purpose of this page is to encourage Shredders to meet new Shredders so that their fan graphs can be extended. The User can click on a Shredder to access his public profile page. The shredders page can seen in figure 4.4 on the facing page.

Shredder, www.shredhub.com/shredder/<id>

This is a page that displays the details for a given Shredder, that has the unique id found in the URL. A list of Shreds that the current shredder has published is displayed in a list view, together with a list of his fanees. The User may choose to challenge this Shredder for a battle, or become a fan of the Shredder. The page is customized to

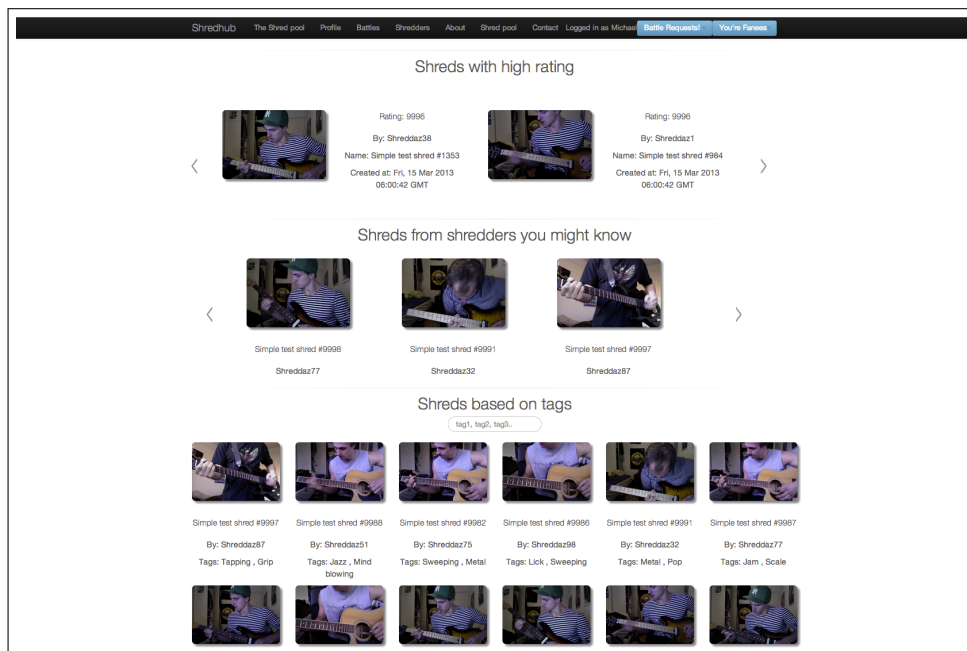


Figure 4.3: The Shredpool (bottom)

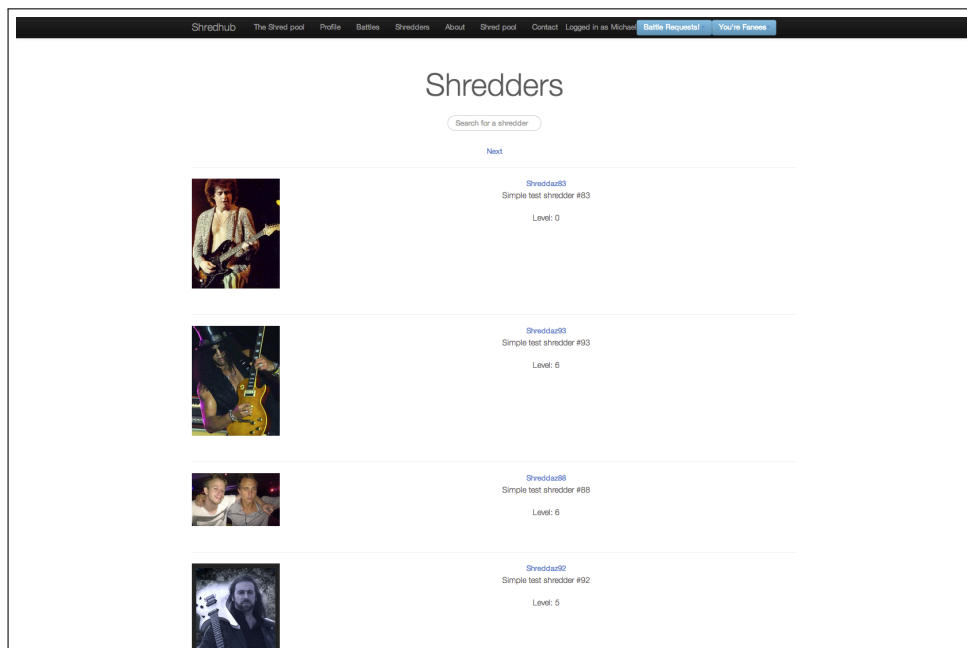


Figure 4.4: The list of Shredders

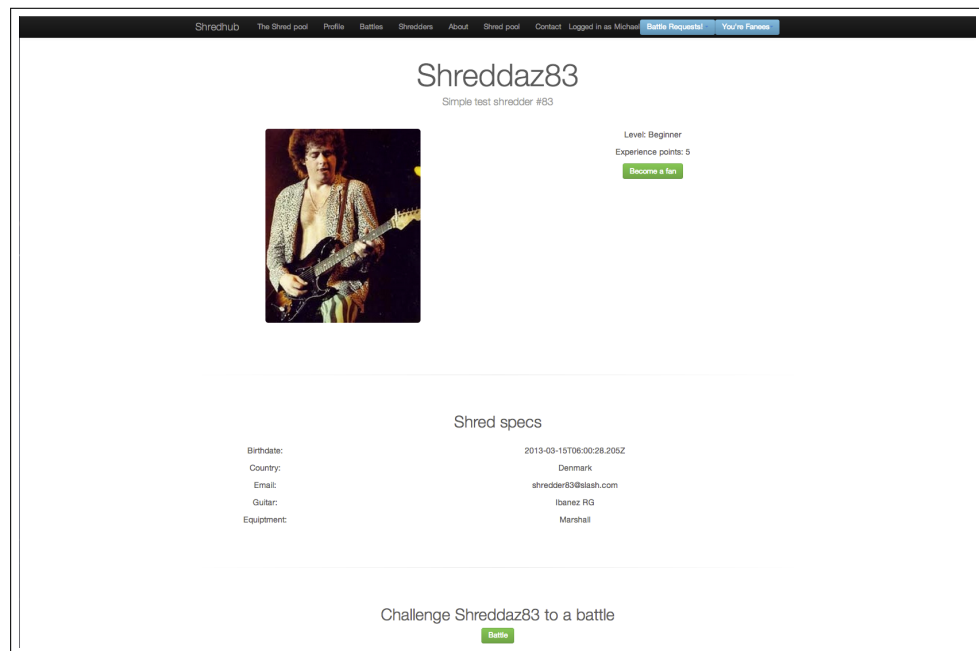


Figure 4.5: A Shredder's profile page

show the relationship the User has with this Shredder. This might be that they already are in a battle, or if a battle request is sent to this Shredder, if they are fans of each other already, and other similar relationships. The page can be seen in figure 4.5.

Battle, www.shredhub.com/battle/<id>

This page displays a battle between two Shredders. If the currently logged in user is one of the battlers, the User is able to upload a Shred for the battle. In a battle I distinguish between the battler who initiates the battle, and the battlee, being the one who is challenged. I have not added an image of a battle, because it won't be discussed in much detail in this thesis.

Shred

A pop-up window displays a particular Shred made by a Shredder. Users can add a rating to the Shred, and add comments for it. The Shred can be accessed from multiple different pages in the app. An example image is given in figure 4.6 on the facing page.

Upload a Shred

For uploading Shreds, a simple pop-up window is displayed so the User can add a Video, a description, and a set of tags. This window can only be accessed inside the Shredpool. This can be seen in figure 4.7 on the next page

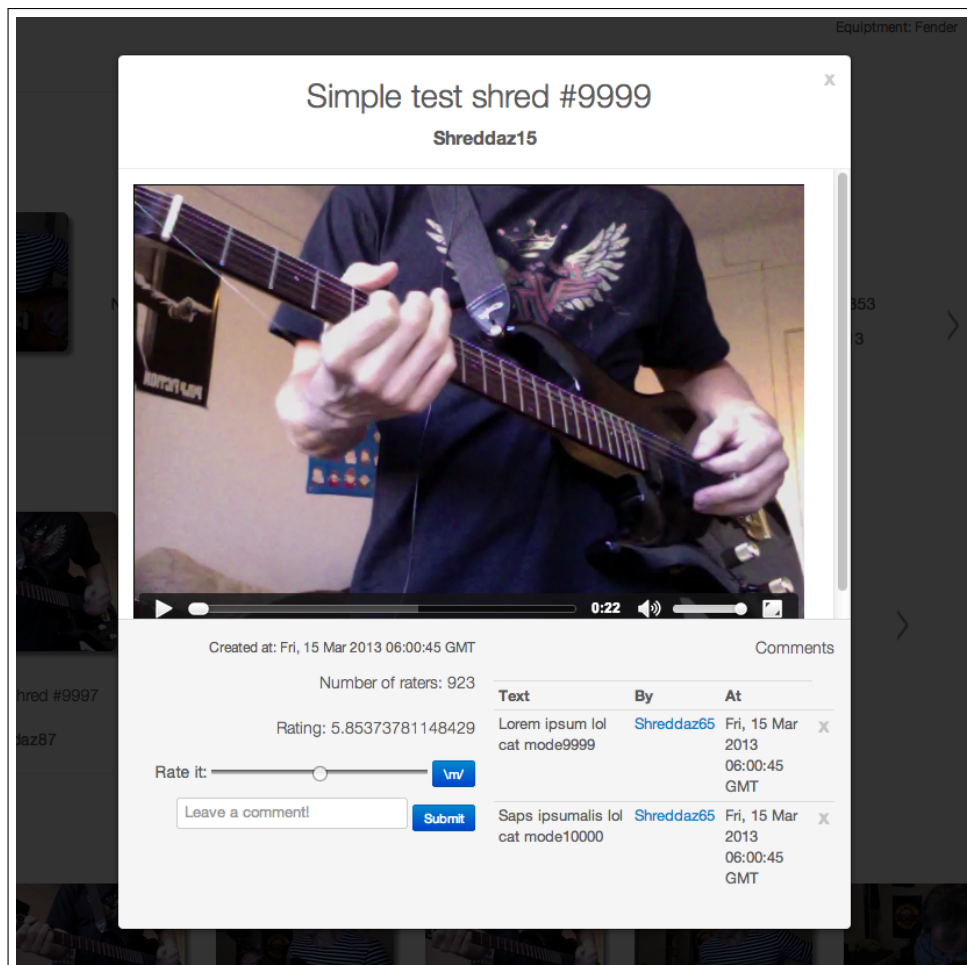


Figure 4.6: A pop-up window displaying a Shred

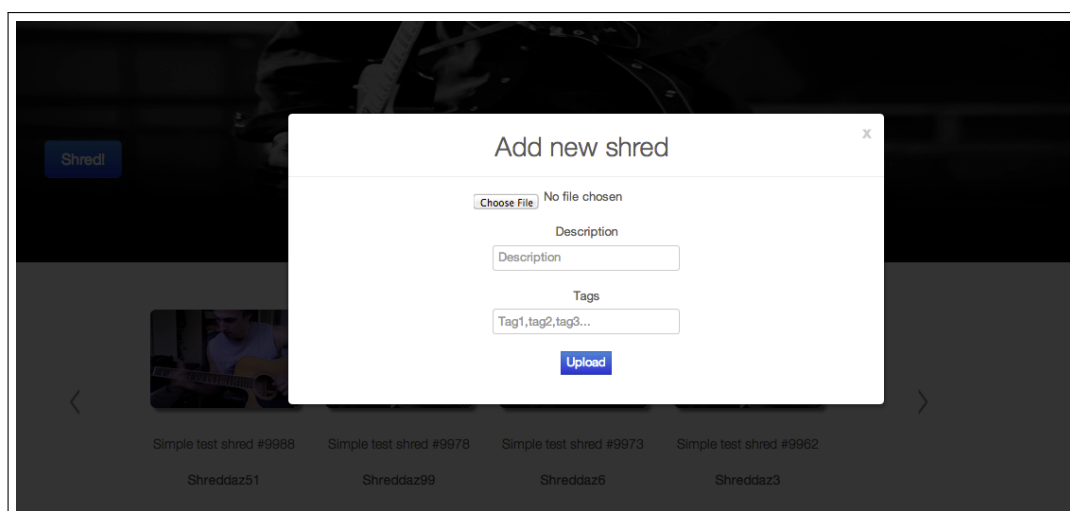


Figure 4.7: A pop-up that lets the User add a new Shred

Chapter 5

Architecture 1.0

5.1 Introduction

In this chapter we look at the architectural details of Architecture 1.0, which is an implementation of Shredhub that conforms to *Reference-model 1.0*. The application is written in Java, and uses the SpringMVC framework. I could have chosen to use another technology like Ruby on Rails, Sinatra for Ruby or a Microsoft .NET framework, considering these are all very popular Web application environments. However, because I happen to know the Java programming language very well, choosing a Java-based Web framework was preferable. Although there are other Java-based Web frameworks in addition to Spring, Spring was chosen because it is very easy to set up, it provides a wide collection of plugin extensions, and most importantly, for the relevance of this thesis, it is one of the most popular Java-based Web frameworks [59].

The application runs on Apache Tomcat, which serves as both a Web server, and an application server. The database is implemented with PostgreSQL. It runs on a database server which for simplicity is deployed on the same physical machine as Tomcat. Here I could also have chosen a different database technology, for instance MySQL or Oracle SQL. However, I chose PostgreSQL because I have experience with the technology, it also has a lot of good and available documentation, and it is a very popular database choice for modern Web applications[33].

In the following sections we look into the implementation details of the source code. We discuss the problems that occurred along the way, choices that were made, and potential alternative solutions. The discussion is divided into three; one part for each software layer in the application.

5.2 Architectural Overview

Architecture 1.0 is a back-end-oriented application. All of the application's logic happens in a Web application that runs on an Apache Tomcat server. In respect to *Reference-model 1.0*, the application is separated into three software layers with different responsibilities; a presentation layer, a domain logic layer, and a data source layer. I have chosen this separation

of concerns in order to facilitate application maintainability and flexibility. I could have chosen to implement everything in two, or even one layer, but this would resolve in classes having very many responsibilities, and tight couplings.

The Web app depends heavily on HTTP sessions to maintain user-state. When a user enters `www.Shredhub.com`, SpringMVC generates an object (called the `HTTPSession` object) who's lifetime lasts throughout the user's session. This object is used as a container for storing state information.

The application's front-end consists of a set of HTML templates that we refer to as **views**. The views are implemented with the JSP template language technology, and are turned into HTML pages by a template rendering engine, provided by Spring. The client user primarily communicates with the application through three different interaction schemes:

1. Hyperlinks
2. HTML forms
3. Buttons or text input-fields that are picked up by JavaScript handlers

For all of the different user-interaction schemes in listing 1 and 2, there will be a corresponding controller handler on the backend. These actions always result in a new view being rendered and returned to the client. Interaction scheme 3 only occurs a few times on the application, in special cases that require highly responsive behavior, in which a server round-trip must be avoided. This is managed by AJAX calls that are implemented inside the views.

5.3 The Presentation Layer

The presentation layer is the first entry point in the application. Its responsibility is to handle client interactions, meaning it will handle authentication, state and session management, and input validation. It is built with the model-view-controller pattern. This decision was made because the structure is familiar to many Web developers, and it neatly separates concerns into coherent and decoupled classes.

5.3.1 Authentication

Users must to be authenticated in order to use any of the pages on Shredhub except the login page. Most of the authentication and access control handling is set up to be handled automatically by Spring. For simplicity, I have chosen to use an HTML form-based authentication mechanism that relies on a username, password and security-role. Other common authentication solutions used in Web apps are HTTP BASIC or HTTP Digest, or HTTP X.509 client certificate exchange. However, I find that form-based authentication fits the simple scope that has been chosen

for authentication in this thesis, and it conforms to *Reference-model 1.0*, because it relies on the server-side sessions.

The form based authentication process works by letting users enter a username and password in an HTML form on the login page. On form submit, the request is picked up by Spring, which will look in the database for a Shredder with the given username, password. The database row for a Shredder also contains a column that represents the Shredder's user-security-role. However, for simplicity there is only one role in this application, which is the one that gives access to everything. If a row with a matching username and password is found, the framework will grant access to the user, and the user will now have access to the whole Web app. To avoid having to re-authenticate for every subsequent request, Spring will behind the scenes maintain a security-context object that is connected to the user's `HTTPSession`. The security-context object simply indicates that the user has successfully logged in once, and is allowed to perform the given request.

5.3.2 State and Session Management

The presentation layer is responsible for managing state associated with a user when he navigates around the application. In Architecture 1.0, this is implemented by using Spring's `HTTPSession` object. Now, considering that Spring offers readily available in-memory objects scoped at session-level, makes it very appropriate to use such objects as caches for data that is frequently accessed. State data is put either in objects that are maintained by the IOC container and scoped at session-level, or directly on the `HTTPSession` object, using a method called `setAttribute(String key, Object value)`. The separation is a matter of separation of concerns; `HTTPSession` objects maintain meta data concerning the user (profile info, battle requests, battles, fanees etc), while other session-scoped objects maintains data regarding the user's page activities (e.g current shred-row, current shredder-page etc).

Data that is used to populate views on the server has to be fetched from the database. Many of these database calls can be avoided if some of the data is stored in memory. The data could for instance be a particular set of Shreds that must be fetched especially quick in order to achieve responsive behavior, user data that is displayed often, e.g the user's name, or a list of battle requests which is meant to be visible on the top navigation-bar at all time. The problem however, is that there is a tradeoff in how much data should be kept in memory, as maintaining too much memory turned out to make the application slow, and in worst case lead to out-of-memory- exceptions. Also, and this is a special case for typical Web 2.0 applications, data tend to change frequently, and users naturally want up-to-date views of the data. Therefore, content regarding the Shreds and the newest Shredders on Shredhub, new fan-connections and other data that frequently changes, for simplicity shouldn't be cached. However, alternative solutions that makes it possible to cache such data is to some extent possible, for example by implementing push-based public-

subscribe service that signals the cache to update whenever an update is made. Or alternatively a pull based solution where some service frequently pulls the database for new updates. Unfortunately, because of lack of time for this thesis, such solutions have not been implemented.

The set of Shreds that are displayed in the shred-pool are partly being cached in memory: the shred-pool is made up of multiple rows of Shreds. Each row consists of 3-5 Shreds, depending on which Shred-row it is, and the user can click a next button in order to change the current row to a new set of Shreds. Now, for each row, the server fetches a set of 20 Shreds from the database, and maintains these in a session-scoped cache. Whenever the user clicks on the “next” button in a row, the server checks the cache for that row to see if the next row of Shreds lies within. If they do, the row is moved one row-size, and this new row of Shreds are displayed. If not, the server fetches another 20 Shreds from the database, puts them in the cache, and displays the first new row. This way, the server avoids many calls to the server, which is very important in order to get quick and responsive behavior when the user clicks the next button. The cache could also be bigger, but then again there is a tradeoff in how big the cache can be without influencing performance.

5.3.3 Input Validation

Input validation is both part of the presentation layer, which addresses form-input rules, and the domain logic layer, which enforces business rules of the input data. In the presentation layer, validation is usually the first thing that happens once a URL request enters the server. Validation is always performed by applying positive filtering, meaning I specify what is allowed, and forbid everything else. Another approach is to do negative filtering where the input is scanned for illegal patterns. However the latter approach is not as secure because it is hard to imagine all possible attack-forms[56]. Also, new forms of attacks might be invented in the future. However, positive filtering has the downside that it might be too restrictive.

The controller handlers and interception filters validate HTML forms, and the controllers often check that the user stored in the session is allowed to perform a given business operation.

5.3.4 Controllers

Controllers are first-class citizens in the presentation layer, who’s responsible for processing URL requests. Controllers are Java classes that are mapped to a specific URL pattern. A simple approach is to have one controller class that is used for every URL supported by the Web app. However, this is not very maintainable, as the class would grow exceptionally large, and have many responsibilities. Other solutions are for instance to have one controller class for every view, or one for every domain object. I have chosen to implement something in between. I chose to implement one controller class for each main resource/domain in the application, in addition to one controller for the home and Shredpool view (simply called

the Logincontroller). In table 5.1 below we can see all the controllers in Architecture 1.0, together with some example controller handlers and their respective responsibilities. Although not displayed in the table, each controller handler is mapped to a unique URL. The controller's main responsibility is to delegate control to a proper business operation in the domain logic layer.

One big advantage with the three-layered architecture becomes clear here; if I choose to change the structure of the controllers, it would not affect the domain logic layer, because the domain logic layer does not depend on the presentation layer. A concrete example of the data flow in the presentation layer is given in section 10.1 in Appendix A.

5.3.5 Views

In this architecture, views are template files that are used to dynamically generate HTML. These views are implemented with the Java Server Pages (JSP) technology. Now, it is considered best practice to avoid implementing business logic in the views[112]. View and Java logic in the controllers together cooperates to create view logic that dynamically generates a page depending on the current state, and user-input. To blend together business logic and view logic implementation would result in tight couplings between two very different concerns. This is one of the reasons I chose to implement the MVC pattern, because it nicely separates these concerns, making it easy to change the view without harming the business logic, and likewise to let the models be unaware of its presentation, so the models and their presentation can change independently.

Main Components

There is one view for each page in Shredhub. These are:

1. The login view
2. The shred-pool view
3. The shredders view
4. The shredder view
5. The Battle view

Also, there are some views that are re-used in the above views:

1. The header view
2. The footer view
3. Show shred view

Each of these views are implemented as a JSP file, e.g login.jsp. The last three views in the list are injected into the other views using special JSP syntax, in order to avoid view duplication. The views contain static

controller handler	Responsibility	view returned
HomeController		
loginPage()	Handles requests for the login page. Fetches the top-rated Shreds from the database and renders the login page	The login view
loginSuccess()	Called when authenticating the user succeeds. Populates the session cache with data fetched from the database	Redirects to the shred-pool
theShredPool()	Fetches from the database the set of Shreds and all the shred-news that are to be displayed in the shred-pool.	The shred-pool view
ShredController		
createShred()	Creates a new Shred and saves it in the database	The shred-pool
postComment()	AJAX supported function that given a shredId and comment-text, adds a new comment to a Shred where the comment-owner is set to the id of the user (stored in session)	None, AJAX request
ShredderController		
getShredders()	Fetches the next page of 20 Shredders that are to be displayed in the list of Shredders view. The page number is maintained in the session object	The shredders view
followShredder()	Adds a new faneer to the user's list of fanees. Updates the session-cached list of fanees for the user	The shredders view
BattleController		
getBattle()	Fetches a battle object given a Battle Id	The battle view
newBattle()	Called when a Shredder accepts a Battle request that is being kept on the session object. Creates a new Battle object. Stores it in the database	The shred-pool view

Table 5.1: The set of controllers in Architecture 1.0

HTML tags that never change, some AJAX functions written in JavaScript, and external links to CSS files and JavaScript libraries. JSP tags are used to inject the domain objects into the view by referencing to the model object that is populated with data in the controllers. The JSP implementation also has some if/else condition tags in order to generate pages given the current state of the application.

When a controller handler finishes, a view is rendered and sent back to the client. Some view logic is also implemented with JavaScript inside the JSP files. In respect to *Reference-model 1.0*, these are just self-contained and independent functions that has one single purpose, and therefore contains no particular architectural structure. An example of how view logic is implemented is given in section 10.2 in Appendix A.

5.3.6 Summary of The Presentation Layer

The presentation layer is built with MVC. Form submits and link actions are picked up by a specific controller handler on the server. The handler performs validation, state management, and delegates to a business function in the domain logic layer.

Models are implemented in a lower layer, but are heavily used by the presentation layer. After a business operation is performed, the controllers are responsible for choosing which model objects to send to a view, which is rendered to HTML and sent back to the client.

The presentation layer depends heavily on the application's state that is implemented in session-scoped objects.

5.4 The Domain Logic Layer

The domain logic layer is the part of the application that receives a specific action from the controller, and performs the necessary business logic needed to complete the action. Now, since the responsibility of the logic layer is to implement the business logic in the application, it is important that this software layer is flexible. Flexibility, means that it can be relatively easy to add new features without harming anything else in the source code, and it should be easy to modify the already existing code. To achieve this, I needed a coherent design, preferably built with design patterns that gives a proper structure to the software architecture. To organize the layer, I have chosen to use the service layer design pattern[43]. With this implementation, each service represents the business operations that operates on a particular resource, or domain in Shredhub (e.g a Battle, a Shredder, or a Shred). However, this is not to be associated with the application's domain objects which concerns the domain resources' data, not functions. Instead, the service abstractions wraps the set of operations that are supported for each resource. Thus, each service class has a set of service functions. The list of service classes with some essential service functions are given below:

1. BattleService

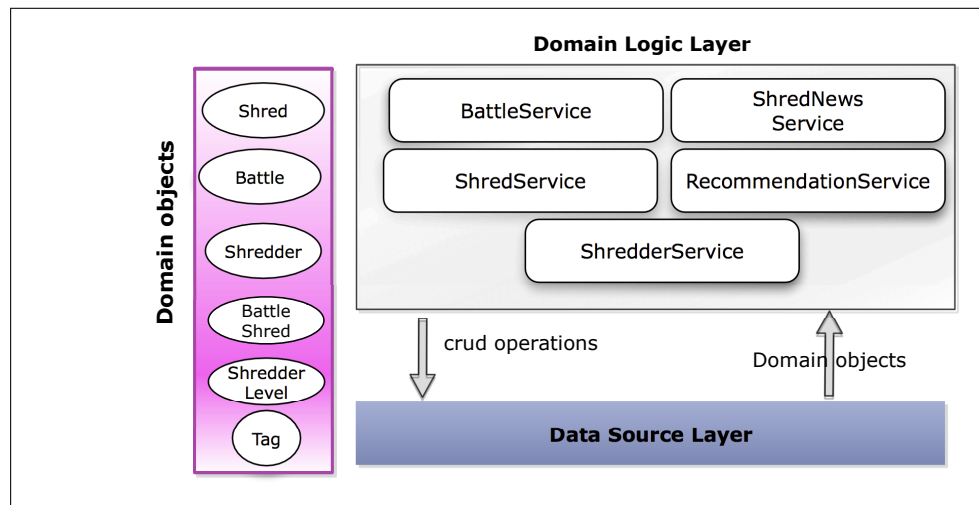


Figure 5.1: The domain logic layer and its connection to the datasource layer

- getBattleWithId
 - getOngoingBattlesForShredderWithId
 - acceptBattleWithId
2. ShredderService
 - addShredder
 - getShredderWithId
 3. ShredNewsService
 - getLatestShredNewsItems
 4. RecommendationService
 - getRecsBasedOnShreddersShredderMightKnow
 5. ShredService
 - getFanShreds
 - getAllTags
 - getShredsForShredderWithId

5.4.1 Service Functions and Domain Objects

An overview of the domain logic layer and its responsibility relative to the data source layer is given in figure 5.1. The figure shows all the service abstractions, and the main domain objects in Architecture 1.0 (there are additional minor domain objects as well, such as ShredComment, ShredRating etc). The domain objects represent the application's core

resources, implemented as simple Java classes with no logical functionality, just attributes with accessor methods. They can be seen to the left on the figure. These objects have references to each other, and are used across all three layers in the application, making them the one means for communicating the domain in the application. The logic layer sends and receives domain objects from the data source layer when they are to be manipulated in the database.

An alternative architecture I could have chosen instead of the service layer pattern, is the transaction script pattern, but this is too simple for this application, and lacks structure. Another alternative is to use the domain model pattern. This is a good approach, but in this case, the domain objects would end up being very big, with lots of responsibility. I prefer the service layer pattern, because it decouples the operations from the object's state.

Most operations in the domain layer follow the same structure; some data is fetched from the datasource layer, this data is manipulated together with the data it got from the calling controller handler, and the result is written back to the database. Also, sometimes the newly updated data is returned back to the controller, so that this can be rendered in a new view. If an error occurred along the way (for instance if illegal data was sent from the controller), an exception is thrown and picked up by the controller so it can return an error view. An example of the data flow in the domain logic layer is given in section 10.3 in Appendix A.

5.4.2 Summary of The Domain Logic Layer

The domain logic layer implements the domain of the application. The layer is divided into two; a service layer that implement business logic operations, and the domain objects which wraps the domain into self-contained data holders. The services forms a facade that is used by the controllers in the presentation layer. The services delegate to the data source layer for persistence.

5.5 The Data Source Layer

The datasource layer is the part of the application that receives a particular CRUD command from the domain logic layer, executes a SQL operation on the database, maps the result to a domain object and returns the result back to the business logic layer. The database is made with PostgreSQL. I have chosen not to use an ORM mapping tool, but rather build Java functions that talks directly to the database using Strings as queries, and mapping query results manually to Java objects. There are many good ORM technologies I could have chosen to use, for example Hibernate, and JPA, which would hide the complexity of serializing Java objects to SQL, and the opposite, and not having to deal with SQL. However, these technologies does not give me the control I need to debug and create flexible marshalling and queries. A tradeoff though, is that this tend to get messy, especially when the queries get many and complicated. I do

however very much enjoy writing SQL queries.

5.5.1 SQL Implementation

The SQL tables that represent the three central domain objects in the application is showed in the example below:

```
CREATE TABLE Shredder (  
  Id          serial PRIMARY KEY,  
  username    varchar(40) NOT NULL UNIQUE,  
  BirthDate   date NOT NULL CHECK (BirthDate > '1900-01-01'),  
  Email       varchar(50) NOT NULL UNIQUE,  
  Password    varchar(10) NOT NULL,  
  Description  text ,  
  Address     text ,  
  TimeCreated timestamp DEFAULT CURRENT_TIMESTAMP,  
  ProfileImage text ,  
  ExperiencePoints int DEFAULT (0) ,  
  ShredderLevel int DEFAULT (1) ,  
  Guitars     text[] ,  
  Equipment   text[]  
);  
  
CREATE TABLE Shred (  
  Id          serial PRIMARY KEY,  
  Description  text ,  
  Owner       serial REFERENCES Shredder(Id) ,  
  TimeCreated timestamp DEFAULT CURRENT_TIMESTAMP,  
  VideoPath   varchar(100) NOT NULL,  
  ShredType   varchar(30) DEFAULT 'normal' CHECK (ShredType = '  
normal' or ShredType = 'battle')  
);  
  
CREATE TABLE Battle (  
  Id          serial PRIMARY KEY,  
  Shredder1   serial REFERENCES Shredder(Id) ,  
  Shredder2   serial REFERENCES Shredder(Id) ,  
  TimeCreated timestamp DEFAULT CURRENT_TIMESTAMP,  
  BattleCategory serial REFERENCES BattleCategory ,  
  Round       int DEFAULT 1 ,  
  Status      varchar(30) DEFAULT 'awaiting' CHECK (Status = '  
accepted' or Status = 'declined' or Status='awaiting');  
);
```

In addition there are many-to-many relations between Shredders and Shreds, Shredders and Battles, and Battles and Shreds (actually BattleShreds, but they are almost identical). It is important to mention them, because it requires the data source layer to perform complex join operations when fetching data from the database.

Now, there are lots of other smaller tables in addition to these (e.g comments, ratings, tags etc), but these are the most essential. To perform the CRUD operations, I have chosen to structure my data tier around the Data Access Object (DAO) pattern. In this pattern, separate DAO objects are responsible for performing the relational data mapping on behalf of a particular domain object (similar to the data mapper pattern). This way the domain objects have no clue on how to persist themselves. An alternative

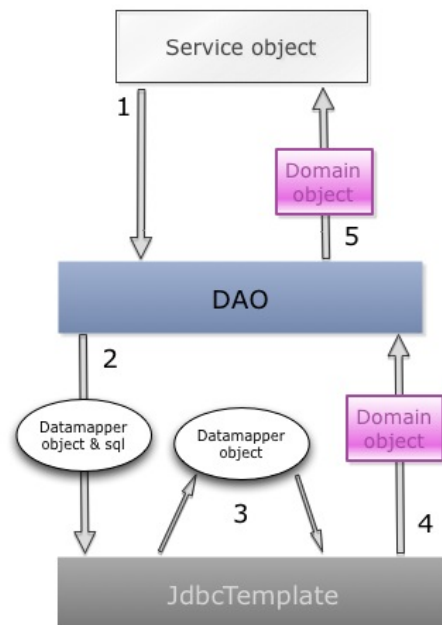


Figure 5.2: Data access pattern in the data tier

to this is to use the Active record design pattern, where each domain object contains persistence code. However I prefer to keep this behavior separated from the domain objects as they would grow quite large and complex if they were to contain all the necessary object relational mapping code. Figure 5.2 shows how object relational mapping is done in the data tier. Here's what happens when the domain logic layer asks the datasource layer to perform a CRUD operation (e.g a Read operation).

1. The logic layer calls a CRUD operation on a particular DAO object, for instance `shredDAO.getShredById(int shredId);`
2. The DAO class uses a JdbcTemplate instance (provided by Spring) to fix boilerplate setups, such as getting, and closing a database connection. The JdbcTemplate is also responsible for executing the database query itself, provided that it gets a SQL statement from its caller. For instance:

```

@Service
public class ShredDAOImpl implements ShredDAO {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public Shred getShredById(int shredId) {
        String sql = "SELECT * FROM Shred s,Shredder sr WHERE s.Owner = sr.Id AND s.Id = ?";
        return jdbcTemplate.queryForObject(sql, new Object[]{shredId}, new ShredMapper());
    }
}
  
```

```
}
```

3. The JDBCTemplate does callbacks to a mapper object provided by the caller. In the above code, the callback calls a function in the ShredMapper class. This class knows how to build a Shred object given the result from a database query that the JDBCTemplate executes. An example of how the ShredMapper class looks like is given below: Example:

```
public class ShredMapper implements RowMapper <Shred>{

    public Shred mapRow(ResultSet rs , int rowNum) throws
        SQLException {
        Shred shred = this.setConcreteShredder();
        shred.setId(rs.getInt("id"));
        shred.setDescription(rs.getString("Description"));
        shred.setOwner( new ShredderMapper().mapRow(rs , rowNum) );
        ;
        // Lots of more mapping ...
        return shred;
    }
}
```

4. The domain object created by the mapper is returned from the JDBCTemplate back to the DAO object, which depending on the type of query might catch an exception to provide nice feedback to the service function. An exception might for example be thrown by the DAO if a Shred with the given Id does not exist.
5. Finally the DAO function returns the domain object back to the domain logic layer.

Clearly, there is a lot of code necessary in order to marshall SQL data. However, the advantage is that the programmer has complete control of how the database results are mapped to Java objects.

5.5.2 How Much Data to Fetch

One issue in the data source layer is to decide how much data to fetch from the database when an object is requested. For instance when a request is made for a Shred, should the DAO function fetch the whole Shred object, fetch the Shred's owner, all the tag objects for the Shred, all the comments and rating etc. This is a tradeoff decision, considering fetching everything requires many SQL joins and much data-mapper processing in Java. These join operations are very performance expensive. But it avoids having to fetch the server for more data at a later point in time, if more of the domain object has to be fetched. One approach is to eagerly fetch every table column and to populate every foreign reference, which would require a large amount of processing, very much data stored in memory, and much data returned back to the client that probably will never be used. The decision I have made is to implement CRUD operations that are

customized for the views in the presentation layer. For example, shredders view need a list of Shredders with their profile data, but without their related list of fanees, shreds and battles. Therefore, the SQL read operation used in this case would not eagerly fetch these other tables, but only the Shredder's profile data. On the other hand, if the shredder view is to be rendered, the database will populate the Shredder with all its fanees and all its Shreds. Thus, I have a lot of customized CRUD operations in the data source layer.

5.6 Summary of The Data Source Layer

The data source layer is responsible for manipulating the database. The database is built with PostgreSQL, where object relational mapping is manually built with Java, instead of using an ORM tool. This requires a lot of Java code, but the code is very flexible and facilitates optimized database marshalling and querying.

5.7 Summary

Architecture 1.0 is a thin-client Java Web app built with a SQL database. All the application's logic happens on the server, where the code is divided into three separate layers; the presentation-, domain logic- and datasource layer. The presentation layer is built with the MVC pattern, in which controller handlers handle HTTP requests, delegates to business operations in the domain logic layer, and upon return, populates a model object with data that is needed to create an HTML page that is sent back to the user. The domain logic layer implements business operations, and delegates persistency handling to the data source layer. The application relies heavily on in-memory Session objects to maintain state. Also, some JavaScript is added to the HTML in order to implement view logic that generates quick and interactive behavior.

Chapter 6

Architecture 2.0

6.1 Introduction

In this chapter we look at the details of Architecture 2.0, which is an implementation of Shredhub that conforms to *Reference-model 2.0*. The application is a thick-client architecture completely implemented with JavaScript, using Node.js on the back-end, and a large-scale JavaScript application that runs in the browser. This way, the back-end is merely a simple interface for manipulating the database. The back-end is made with two popular NoSQL databases. Redis, for authenticating users, and MongoDB for persisting the application's domain. The front-end uses various third-party frameworks that expands the JavaScript programming language. These are Backbone.js for code-structure, AMD for dependency management, and JQuery for cross-browser DOM manipulation. For simplicity in this chapter, we will use the term *App* to refer to the JavaScript application that runs in the client's browser, and we will use the term *API* to refer to the code that runs on the back-end.

6.2 Architectural Overview

The front-end is a large-scale JavaScript application. Now, as discussed previously, building large JavaScript applications is difficult, primarily because it lacks programming language idioms like classes, namespaces and dependency handling. To get a modular and flexible codebase for this application, a solution to the aforesaid issues is to use open-source frameworks that provide module features and dependency handling. In addition the application is built using the MV* pattern. The pattern fits the requirements for this interactive Web app mainly because it separates the domain logic from the view logic, such that these concerns can be implemented independently. Also, I am free to decide how to implement controller logic. I have chosen to divide this concern into two parts; a Router module which handle requests for the main pages on Shredhub (coarse-grained requests), and views, which handle finer-grained requests for minor user interactions. I could have designed the *App* around a traditional MVC architecture, but this wouldn't give me

an intuitive controller separation, because all controllers are treated equal in this pattern. The MVVM pattern would also have been a fine choice, because Shreds and Shredders are displayed in multiple ways, and thus each graphical representation would be implemented in a separate view-model object. However, this design is a bit more complex than MV*.

The *App* is composed of a set of loosely coupled modules, where each module contains a set of zero to many **models**, **collections** and **views**. These are core entities in the application that together provide domain data, business operations, controller handling and view logic. In addition, there is a **Session** module which offers a facade to manage session data, and a **Router** for navigating between pages. Lastly there is the **Mediator** which is a module that coordinates communication between views and models. Each module is a separate JavaScript source file. The Asynchronous Module Definition pattern is used to define dependencies between each module.

The API is organized as a Rest API[40], where the first-order citizens are the application's domain objects. In Architecture 2.0, these are Shreds, Shredders, Battles and BattleRequests. Thus, the API offers a set of self-contained operations that manipulates these resources. In respect to *Reference-model 2.0*, the API is stateless. To achieve this, every Rest operation contains all necessary state information.

The reason Node.js was chosen on the back-end, is primarily because it uses JavaScript. This way, JavaScript is the one and only programming language used through the whole application. Now, because both databases in Architecture 2.0, the API, and also the *App* communicates with the same data-format, JSON, no marshalling has to be done. This does simplify the programming model. Figure 6.1 on the facing page shows an overview of the main components in Architecture 2.0. The figure doesn't include the Mediator, or Session, but they are separate modules in the browser.

In the rest of this chapter we go into the implementation details of Architecture 2.0. The following text is divided into a front-end, and a back-end section.

6.3 The Front-end

Also referred to as the *App*, the front-end is composed of a large JavaScript codebase and a set of HTML template files that are used to dynamically generate HTML.

6.3.1 The Bootstrapping Process

In Architecture 2.0, a fairly large JavaScript application has to be downloaded and initialized in the client's browser during the client's initial request to the Shredhub. I call this the bootstrapping process, because the client will ask for a small HTML page that contains one single line of JavaScript. This statement is responsible for starting a recursive process that

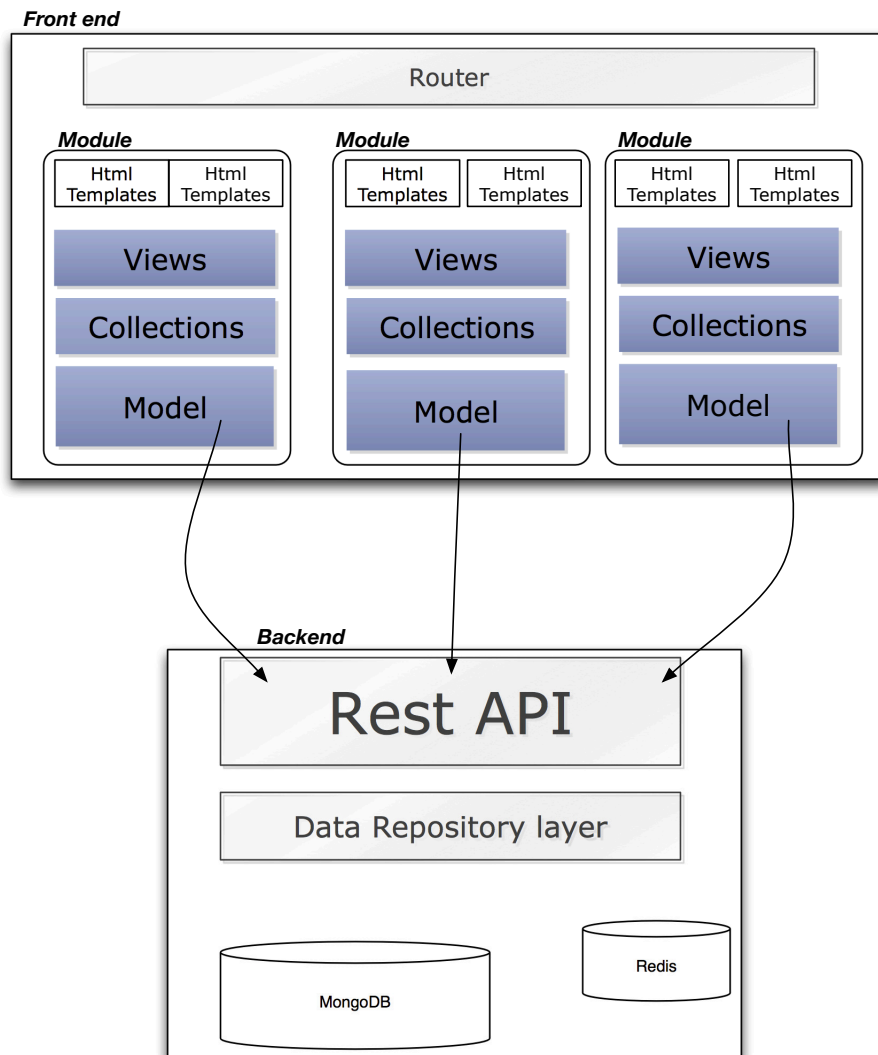


Figure 6.1: The main software components of Architecture 2.0.

loads in the rest of the *App* from the server. In detail, the bootstrap process works like this:

1. The client visits www.shredhub.com and the server responds with a file called `index.HTML`
2. `index.HTML` contains the line
`<script data-main="/app/config" src="/vendor/js/libs/require.js"></script>`,
which will fetch a JavaScript file called `require.js` from the server
3. `require.js` is a framework that implements the AMD specification. It will access a module called `/app/config` (outlined in the script statement above), which contains a reference to the **main** function.
4. The JavaScript file that has the main function is fetched from the

server, and the main function is called.

5. The main function is responsible for instantiating objects that will be globally accessible (that is, accessible through the whole codebase). This includes the Session, Mediator and the Router object. Also, a globally accessible object called *app* is created. This object will cache HTML templates in the browser's JavaScript heap memory, so that HTML templates won't have to be fetched more than once.
6. When the Router object is initialized it will start listening to URL changes.
7. At the end of the bootstrapping process, the Router will handle a request for the home page. This will result in the Home page view being created and rendered in the browser.

One thing to mention is the decision chosen for how to fetch JavaScript files and HTML template files. In chapter 3, we discussed two ways of doing this, either lazily, or eagerly (all at once). At first, I went with a lazy loading approach, where templates and JavaScript files were fetched only when needed. However because these files are many and small-sized, and the browser sets up one HTTP connection for every file, it lead to a lot of unnecessary HTTP round-trips. Therefore, I chose to merge all the JavaScript files and HTML templates into one single JavaScript file. In addition I have compressed the file in order to minimize the initial fetch of the *App*. The result was much better, because the browser would never have to ask for JavaScript or HTML resources after the initial phase. However, had the code base been significantly larger, this approach would possibly not have been an optimal solution.

6.3.2 Router

The Router is the component that organizes routing between the Shredhub's main pages. Normally, triggering a hyperlink in a Web page would make the browser send the request directly to the server. This however, is unwanted in Architecture 2.0, because the front-end is supposed to decide when and how to contact the server. This is where the Router comes in. The Router is configured to listen to every hyperlink-event so that when such an event is triggered, the Router is notified, and it will call the **event.preventDefault()** function on the browser, which in effect tells the browser not to issue the URL request to the server. This way the Router has **hijacked** the request, and is now able to decide what will happen.

To some extend, the Router works as a controller from *Reference-model 1.0*, in that it receives a particular page request (for example www.shredhub.com/shredders), and performs the necessary work to handle the request. In Shredhub, there are five different hyperlink possibilities; one for each of the five main pages. Thus there will be five controller handlers, or routes, as they are called in Architecture 2.0. After the router has hijacked a URL request from the browser, it will call the route

handler for that particular URL. A URL-handler mapping is configured in a file called router.js. It looks like this:

1. 'shredPool': 'renderShredPoolView', *//www.shredhub.com/shredpool*
2. 'shredder/:Id': 'renderShredderView', *//www.shredhub.com/shredder/<shredderId>*
3. 'shredders': 'renderShreddersView', *//www.shredhub.com/shredders*
4. 'battles/:Id': 'renderBattleView', *//www.shredhub.com/battles/<battleId>*
5. '*actions': 'renderHomeView' *//www.shredhub.com*

6.3.3 Models

The models represent the domain resources of the application, which implement both business logic and data attributes. Hence they implement the Domain Model design pattern. I chose this as opposed to having a separate service layer. An additional service layer does result in more decoupling and separation of concerns (business operations and data holders in this case), but it also leads to more code and additional source code files. In this architecture, less code and files are to some extent preferable, considering these are data that must be transmitted over HTTP.

Models also use the Active Record design pattern, meaning they are responsible for knowing how to perform CRUD operations on themselves. Thereby avoiding additional modules that concerns only the data source handling. Now, because the database lives on another physical machine, CRUD'ing happens via HTTP. This is done with AJAX, so that API communication happens asynchronously, and in effect won't block when data is needed from the back-end. An example of a business operation in the model is showed in section 11.1 in Appendix B.

6.3.4 Collections

Considering that the application has many "collections" of models, for example a list of Shredders on the Shredders page, multiple rows of Shreds in the Shred-pool page etc, it makes sense to encapsulate these models in separate modules (collections). This way, a collection is a container for multiple coherent models. The motivation for this, is that the collections can also work as Active Records, in that they can be responsible for fetching and maintaining a particular set of Shreds or Shredders from the database, regarding the collection of models they control. For example a Shred-Collection representing a row of top-rated Shreds, would know how to fetch the top-rated Shreds from the API.

6.3.5 Views

The set of pages in Shredhub are in Architecture 2.0 separated into logical coherent views. These views are JavaScript objects that hold a reference to one or more HTML templates that it is responsible for maintaining. This

means the view handles all the user-interactions that happens inside the HTML it represents. A View contains zero or more model and Collection objects, such that it knows how to visualize these domain objects. Also, the models are used to delegate business operations to. The view's main job is to render its HTML template(s) together with its containing set of collections and/or models. In addition, the view is responsible for maintaining state for the particular HTML portion of a page it represents. A view can contain one or more sub-views, such that views can form a tree of views. Views are created either by the Router when a page is to be rendered, or by a parent view, when it needs to create a child view that will render a smaller part of HTML inside the current view. Views are deleted and added whenever a new page is to be rendered in Shredhub, and also when minor parts of a page is to be rendered. The set of views in Architecture 2.0 is given in table 6.1.

View Name	Responsibility
Scaffolding View	Contains the header and footer that is contained in every page. Always wraps one sub-view
Home View	Represents the login page. Wraps a set of sub-views; a list of ShredThumbnail Views, and a ShredModal View
Shredpool View	Represents the Shredpool page. Wraps a set of ShredRow Views and a ShredModal View
Shredders View	Represents the list of Shredders page
Shredder View	Represents the Shredder page
ShredRow View	Represents a particular row of Shred thumbnails. Maintains state for the row, so that it knows when to advance to a new row in the same collection of Shreds. Each column in a row is wrapped in a ShredThumbnail View
ShredThumbnail View	Represents a Shred thumbnail, consisting of a thumbnail image, and metadata about the Shred. Notifies the mediator if the Shred was clicked, in order to tell it to open a ShredModal View to play the Shred itself
ShredModalView	Represents the popup window that plays a Shred. Handles user events like rate button clicked and comment text submitted

Table 6.1: The set of views that are implemented in Architecture 2.0

Event handling

Each view is set up to listen to certain events that are relevant to that view. For example a ShredRowView is initialized to listen to the *next-row* button clicked, and a ShredModal view is initialized to listen to the *rate* button. In

each view, there is an **event handler** function for every event it listens to. In Architecture 2.0 I have separated the types of events into two: **View logic events** and **Domain logic events**. A view logic event is something that simply manipulates the DOM tree in order to alter the HTML. A Domain logic event however is more like a controller handler from Architecture 1.0, where the event requires some business operation to be executed.

When a view is to be deleted (in favor for some other view to be rendered “over it”), all its DOM elements must be removed. Also, it is especially important to remove any event listeners the view has registered. If they are not removed, they will continue to exist and listen to events, so that if a view is recreated, its old events will co-exist with the newly created event listeners. Now, when an event is triggered, there might be multiple listeners listening to that event, and in affect calls to the same handler function, so that it is executed more then once. The result could be multiple equal write operations sent to the database. Also, this could lead to slow performance, because the listeners consumes memory. This is a problem that often occurred during the implementation of Architecture 2.0, especially because many of the views in Shredhub have multiple child views. The solution was to implement a recursive remove function that is called on a view and all its children whenever a view is to be removed. This function removes the DOM elements for the view, and deregisters all its event listeners. An example of how a view is implemented can be found in section 11.2 in Appendix B.

HTML Templates

Each view knows where in the DOM tree to put the particular HTML template(s) it is responsible for. For example the ShredRowView that represents the ShredRow of top-rated Shreds holds an HTML template called *ShredsRow_topRated.HTML*, which the view will inject into the HTML tag `<div id="topShreds"></div>`. Just like JSP template files in Architecture 1.0, the templates in Architecture 2.0 are not pure HTML files, but contains special syntax that can reference model data, and supports loop statements, conditional statements and other simple programming language statements. However, there is a big difference between the way I have implemented templates in Architecture 1.0 from Architecture 2.0. In Architecture 1.0, the templates were coarse grained, and contained a lot of view logic to decide the outcome of the HTML. In Architecture 2.0, I have decided to create many, and smaller fine-grained HTML templates, and factorize out as much view logic as possible into the view. This is often done by letting views have references to multiple fine-grained HTML templates. These all have the advantage of being able to be reused in other parts of the app. Also, I have implemented a couple of fine-grained views, that are being reused across the app. For example, a ShredThumbnailView is reused as a child view of other views who need to display Shred thumbnails.

Abstracting view logic out of the HTML templates and into the views, facilitates a better decoupling of HTML markup and view logic. This

decoupling was not achieved in Architecture 1.0. One example: In Architecture 1.0, the `ShredderView.JSP` contained many if-checks to figure out the relationship the user had with the particular Shredder that was to be displayed. A unique HTML output was to be created depending on:

- if the visited Shredder is actually the same Shredder as the user
- else if the user has sent the Shredder a battle request
- else if a battle request from that Shredder is currently pending
- else if they are currently in a battle
- else; the user should then challenge the Shredder to a Battle

Therefore, the JSP template had to include HTML markup for every possible outcome, and depend on complex JSP if-conditions to know which part of the HTML to render (together with the rest of the JSP page of course!). In Architecture 2.0, all of this is figured out **before** the rendering process begins. Now, the HTML for displaying each of these five different Shredder relationships are represented in separate (fine-grained) HTML template files. This way, when the rendering process begins, the view will pick the proper HTML template depending on the result of the if-check, and inject this template into the DOM. In effect, the templates contain very little view logic, only enough to display the data from a model object it receives when the HTML is to be rendered.

6.3.6 The Mediator

There are many cases in which disparate components need to communicate with each other in the *App*. For instance, separate views need to communicate with each other, and sometimes views need to communicate with model objects they don't necessarily have direct references to. In order to facilitate a loosely coupled, flexible and efficient communication model, I have chosen to use the Mediator design pattern [47, p. 305]. This is a component where views and models can publish and subscribe to certain events, such that when someone publishes to the Mediator that an event has happened, the Mediator will notify every listening entity (subscriber), and call all of the handler functions the subscribers have registered with the Mediator. This solves the need to have many object references in every view and model in order to call functions across the objects.

6.3.7 Session

In Architecture 2.0, state is completely implemented on the client so that the server has no awareness of any logged-in user or session. To do this, the *App* must have a way of storing and manipulating state data on the client. This could be done by storing data in the browsers JavaScript memory, considering there is never necessary to do a page refresh, meaning the JavaScript heap will never be flushed. Unfortunately, this could negatively

affect the browser's performance if the data size grows quite large, and also, if the user would happen to manually refresh the page, the JavaScript memory is cleared. It could also be done by storing all the state inside cookies, but this is not as secure considering the state data would need to be included in every HTTP request. This of course, would also waste and consume very much bandwidth. The solution chosen is to use HTML5 WebStorage, which neither affects browser performance, or is subject to data loss on page refreshes. The storage size is big enough to hold many megabytes of data (depends on the browser), so in practice there is no need to limit how much user data to store in the browser. I have chosen to use session storage and not local storage, so that state data is restricted to a session. This is because the data I store in Web storage is naturally bound to a "session", and shouldn't last for any longer than this. There is one misfortune with this design decision however; some old browsers do not implement HTML5 Web Storage. Now, I have not have the time to design a backup solution for such users, however a simple approach is to check during the bootstrap process if the current browser supports Web Storage, and if not, use the browser's JavaScript memory or cookies to store state data.

The *App* uses `sessionStorage` to store user data only, considering much of the other state data is maintained in the views (i.e JavaScript memory). The storage is populated with user data when the user is authenticated. This data includes:

- User profile data, such as username, address, birthdate, list of guitars etc
- Authentication details (a token made up of username and password)
- List of the user's fanees
- List of the user's current sent and pending battle requests
- List of the user's current battles

The Session module mentioned previously is a facade that wraps the browser's session storage API.

6.3.8 Summary of The Front-end

The front-end in Architecture 2.0 is a large-scale JavaScript application that is loaded into the browser when the user first accesses Shredhub. A special module is configured to "hijack" hyperlink events in order to avoid that the browser automatically sends requests to the server. Instead, every user action is handled in the front-end code.

In addition, state and session handling is completely handled in the front-end, and server communication is only done via AJAX calls. This way, browser refreshes will never occur.

6.4 The Back-end

6.4.1 The Rest API

The Rest API is the communication boundary between the *App* and the server. The back-end exposes all of its available operations through the Rest interface. For each domain object, there are four type of operations, one for each HTTP method: *Get*, *Post*, *Put* and *Delete*. Now, in order to offer more complex operations then just a combination of a resource and an HTTP method (e.g *Get* + *Shred Id*), the Rest API adds an additional verb that describes a specific operation that is to be performed. An important property of the Rest operations is that they are self-contained, in that they have all the state information needed to perform the operation. Take for example the following URL:

GET:api/shreds/NewShredsFromFanees/5142b8fc174328d087ac49b9/?offset=20&page=3

The long string represents a unique Id (uid) for a Shredder. With this request the back-end will query the database for a set of Shreds that are made by the Shredder with the given uid's fanees. The returned list is a set from the query result, starting at result number 3*20, and the size of the result being 20 Shreds. In addition to the URL, the HTTP request contains an authentication token that the API uses to verify that the user is allowed to perform the operation. The *App* appends this token to the HTTP Authorization header parameter in every API request. In this example, the API would fetch the user that is given in the authentication token from the database, and verify he has the exact same user id as the one given in the URL. If so, the API operation is executed. In a similar operation in Architecture 1.0, the back-end would look at the Session object that's in memory to get the user who issued the request, and by knowing what page number the user is currently at (also stored in a session object), and the amount of Shreds that are displayed on the current page, the back-end would have all necessary information to issue the request. Thus all the necessary information in that case is on the server, while in Architecture 2.0, all necessary information is in the HTTP request.

Another example is when a new domain object is to be stored in the database. In this case, a raw JSON object is sent to the API, which would put the data (HTTP payload) directly in the database without any marshalling:

```
Request URL:HTTP://localhost:3000/api/shreds
Request Method:POST
Content-Type:application/JSON
Request Payload
{"description":"Sweet Shred in C-minor",
"shredRating":
{
  "numberOfRaters":0,
  "currentRating":0
},
"shredComments":[] ,
```

```

"owner" :
{
  "_id" : "5142b8fc174328d087ac49b9" ,
  "username" : "Michael"
},
"tags" : [ "Scale" , "Speed-picking" , "Melodic" ] ,
"shredType" : "normal" ,
"timeCreated" : "2013-03-18T12:24:13.363Z" ,
}

```

Every Rest operation returns with a status code, indicating if all went well, in addition to the result from the database query. The status code is one of the HTTP status codes which serves to inform the client if the operation was successfully executed or not. The HTTP status codes used are:

- 200 OK, meaning the operation was performed, and the response contains JSON data
- 400 Bad Request, meaning the user tried to perform an operation with illegal input parameters. An example is if the user tried to add a rating to a Shred with a value higher than 10.
- 401 Unauthorized, meaning the user is not allowed to issue this request. An example is if the user tried to add a Shred, and the owner is set to reference a Shredder who's un-equal to the Shredder identified in the authentication header.

There are many other status codes supported by HTTP, which I could have used in order to enrich the error messages used in the application. However, this goes a bit out of scope for this thesis. The point here is to show how error handling can be done in a stateless and decoupled fashion; the back-end does not know how the *App* treats the error message. This is apposed to Architecture 1.0, where in cases of an error, the server will return a completely rendered HTML error page back to the client.

6.4.2 The Data Repository Layer

The data repository layer is the part of the back-end that implements the Rest API and communicates with the database. It is organized as a set of controller modules; one for each domain resource. Much like controllers in Architecture 1.0, the controllers in Architecture 2.0 are mapped to a specific URL. However, instead of going through a complex domain logic layer, and data source layer, the controller's responsibility is much simpler. Most importantly, it doesn't generate views, just pure JSON data. Generally a controller handler does:

1. Validate the parameters given in the URL query string, request body and authentication header.
2. If there is illegal input, send a proper HTTP status code back to the client.

3. If not, create a database query with the URL arguments and request body and execute a query on the database.
4. Send the result (without marshalling) back to the client.

6.4.3 Authentication

Authentication in Architecture 2.0 is implemented with the HTTP Basic Access Authentication protocol[46]. The reason this was chosen is because it conforms to *Reference-model 2.0*, where the server must be stateless, and HTTP basic auth does not rely on any session or cookie. The *App* authenticates users through the Rest API by concatenating the user's username and password into a base64 encoded string. This string is appended to the HTTP authentication header parameter, and is sent with every API operation (except the initial request for the home page).

HTTP basic access authentication is not a very complex, and especially not secure protocol, considering the data is not encrypted. A first improvement would therefore be to enforce the use of HTTPS in order to properly encrypt the username and password. Other authentication protocols could also have been chosen. One popular solution is OAuth[91], which is much used in Web 2.0 applications. However, this is a somewhat complex protocol that requires some effort to implement.

6.4.4 The Databases

There are two databases used in Architecture 2.0. The reason for this is because I have two different persistency needs. One is to persist the domain model in a flexible and efficient way, which is done with MongoDB. The other is to have authentication data available in a highly efficient manner, which is done with Redis.

User Authentication With Redis

In Architecture 2.0, the authentication token needs to be verified in every URL request except those regarding the home page. Therefore, the back-end must have a highly efficient way to authenticate each API request. By using Redis, I store two key-value pairs for each user, one that maps a username to a unique Id, and the other maps the unique Id to the password that belongs to that user. The unique Id is the same unique Id that is used for that particular user in MongoDB. An example of a user in Redis looks like this (The long string represents a unique Id):

```
username:Michael:uid 5142b8fc174328d087ac49b9
uid:5142b8fc174328d087ac49b9:password 1234
```

Keys are on the left-hand side of the white space, while values are on the right. The colons are used to infer a descriptive semantic. For example the key *username:michael:uid* describes the value *unique id for an entity with*

username equal to "Michael". A similar semantic applies to the second key-value pair. In order to authenticate a user, the backend does the following lookup (in pseudocode):

```
function authenticateUser(username, password) {  
  
    // Create a string on the form 'username:<username>:uid':  
    usernameStr = 'username:' + username + ':uid'  
    get the value with key=usernameStr from Redis, put result in res  
  
    if ( success ) {  
        // A user exists with the given username. Now check the password  
        // Create a string on the form 'uid:<uid>:password'  
        var uid = res.toString();  
        var passwordStr = "uid:"+uid + ":password"  
        get the value with key=passwordStr from Redis, put result in res  
  
        if ( success ) {  
            if ( password === res.toString() ) {  
                // Correct password was given. Return success together with the uid  
            }  
        }  
        // Authentication failed, return proper error message  
    }  
}
```

The uid is returned so that it can be used to fetch the newly authenticated Shredder from MongoDB.

The reason Redis was chosen is because of its extremely high speed when it comes to simple key-value pair lookups. Redis is not meant for complex and structured data, but is specialized to operate on simple HashMap data structures. Also it favors speed over durability, something that is preferable in this occasion, considering the only time I perform write operations to Redis is when new Shredders are created. At this point, I eagerly write a snapshot to disk in order to force durability.

Other alternative solutions could have been to use for example Riak, or Voldermort. However, these favor distribution and very high availability over speed, which is the reason why I chose Redis instead.

MongoDB

The domain in Architecture 2.0 is persisted using MongoDB. The reason I chose MongoDB for this, is mainly because it uses a JSON-like format to persist data, which is a very nice fit for the domain; much of the domain in Shredhub can be modeled as a nested structure, which is very appropriate to implement with JSON. This nested data structure is very typical Web 2.0 applications that have blog-posts and comments (with commenters). Also, considering the MongoDB database can be manipulated directly using JavaScript, there is no need to implement additional data mappers for creating queries and marshalling of query results. A final reason I chose MongoDB is because of MongoDB's schema-less document model, allows for highly flexible data modeling solutions. Thus, I can very

easily customize my MongoDB documents to fit the data exactly like they are displayed in the *App*. This does require some duplication of data, but it does avoid relations across the documents, that normally requires join operations in order to fetch the necessary data. Another compelling NoSQL solution is to use CouchDB, which also support direct database manipulation with JavaScript. However, I went with MongoDB mainly because it's probably the most popular NoSQL database as of 2013[83].

An example of the set of MongoDB collections implemented in Architecture 2.0 is given below:

```
// Shredder
"_id" : ObjectId("5142b8fc174328d087ac49f7"),
"username" : "Shredder64",
"fanees" : [
  {
    "_id" : ObjectId("5142b8fc174328d087ac49f5"),
    "username" : "Shredder62",
    "profileImagePath" : "shredder62profile.jpg"
  }
],
"birthdate" : ISODate("2013-03-15T06:00:28.202Z"),
"country" : "Denmark",
"profileImagePath" : "shredder64profile.jpg",
"email" : "shredder64@htomails.com",
"guitars" : [
  "Gibson flying v"
],
"equipment" : [
  "Marshall JCM 800"
],
"description" : "Simple test shredder #64",
"timeCreated" : ISODate("2013-03-15T06:00:28.202Z"),
"shredderLevel" : 84
```

```
// Shred
"_id" : ObjectId("5142b90a174328d087ac4a2e"),
"description" : "Simple test shred #19",
"owner" : {
  "_id" : ObjectId("5142b8fc174328d087ac49c4"),
  "username" : "Shredder13",
  "imgPath" : "Shredder13Image.jpeg"
},
"timeCreated" : ISODate("2013-03-15T06:00:42.313Z"),
"shredType" : "normal",
"shredComments" : [
  {
    "timeCreated" : ISODate("2013-03-15T06:00:42.313Z"),
    "text" : "This is a very nice Shred!",
    "commenterId" : ObjectId("5142b8fc174328d087ac49ea"),
    "commenterName" : "Shredder51"
  }
],
"shredRating" : {
  "numberOfRaters" : 946,
  "currentRating" : 8188
}
```

```

},
"videoPath" : "shred11234.mp4",
"videoThumbnail" : "shred11234_thumb.jpg",
"tags" : [
  "Fast",
  "Sweeping",
  "Tapping"
]

```

Similarly there are collections for Battles and BattleRequests. An example of how a Shred is fetched from MongoDB on the back-end is given below:

```

exports.getShred = function(id){
  Shred.findById(id, function (err, shred) {
    return shred;
  });
}

```

Notice no marshalling needs to be done, because the object that is fetched is simply a JSON object.

Discussing MongoDB over SQL for Shredhub

Notice in the example above, there are only 4 different MongoDB collections. This can be compared with the SQL implementation from Architecture 1.0 that is implemented with 16 tables. The reason I have chosen to limit the amount of collections as much as possible is to avoid the tedious join operations that would normally be needed in SQL. Joins are very slow, and also, they are not natively supported in MongoDB. One has to manually implement joins by performing multiple subsequent read operations across documents. My solution however emphasizes the use of duplicating data so that documents fit the domain in the way they are visualized in the *App*. Look for example at the Shred document in the example above. The owner consists of his Id, username and image path. Also, the comments contain the comment-owner's Id and username. This is exactly enough data that is necessary in the *App*, in order to visualize the Shred. In a normalized SQL (i.e Architecture 1.0) implementation the owner would just be represented by a foreign key, and during a Shred-fetch a join operation would have to be done for the Shred-owner, all the comment owners, every tag, and every rating.

One misfortune with this design, however, is if any of the duplicated values were to change in the original document. For example if the Shredder with name Shredder13 was to change his profile image. In this case this update would have to be propagated to every place in the database where that particular image is referenced. However, I have acknowledged this fact simply because profile images aren't something that is likely to change very often. Another misfortune is that the database is somewhat *App*-aware. Imagine the API was to be used by other clients, maybe third party clients that would have other requirements to the amount of data that is populated with a particular fetch operation. One could argue that this customized duplication of data is somewhat

specialized and enclosed for future needs. However, I state that this data modeling decision is still very flexible, considering every domain resource always include their uid, making it possible to force join operations if more data needs to be populated in a given query.

A big advantage with the MongoDB design however, is that no mapping needs to be done when objects are fetched and stored in the database. This reduces the amount of boilerplate code needed, and simplifies the whole programming environment both on the back-end and front-end, because only one common data structure is used. Consider for example the difference between the `getShred` function above, and the `getShredById` function in Architecture 1.0, which required a separate `ShredMapper` class. Now, the latter could be avoided by using an ORM tool, but there is still a lot of boilerplate code with that solution, it is just hidden in a separate third party code package.

In addition, many benchmarks[127][11][100][12] show that MongoDB performs faster then various SQL databases. However, a study for benchmarking database performance in a social network application shows that MongoDB and a SQL implementation performs rather equal, but a SQL solution mixed with a caching system (Memcached[76]) is superior[15].

6.4.5 Summary of The Back-end

The back-end is built as a Rest API that exposes a set of self-contained and stateless operations.

The API uses an authentication mechanism that does not rely on server-side state. Therefore, every API request must be authenticated. This is done very fast by using a key-value database that is kept in memory.

MongoDB was chosen to persist the domain objects, because it avoids having to do marshalling, and does not enforce any structure, so that data can be stored exactly like it should look like in the user interface. This requires some duplication of data in the database. A potential drawback with this design is that the data is somewhat rigid.

6.5 Summary

In this chapter we have looked at Architecture 2.0, a thick-client JavaScript Web app built with NoSQL technologies. The responsibility of the back-end is rather simple, merely to serve as a central data-repository layer for the database, and to send the front-end application to the browser on initial requests.

The front-end application has a decoupled structure where the code is organized into coherent modules. The front-end avoids letting the browser automatically contact the server, by hijacking hyperlink events. It is responsible for maintaining state, and perform business logic operations.

The back-end is built around a Rest API that exposes self-contained operations to client users, and it communicates fine-grained objects instead

of HTML pages.

Chapter 7

Performance and Source Code Analysis

7.1 Introduction

A major goal for any interactive Web application is to minimize the response time for user actions, because it has a significant impact on the user-experience. users most often do not have patience to sit around and wait for slow page requests, which in some cases could result in users abandoning the site in favor of other competitors. The response time for such an action is a measurement of the time it takes from when the user initiates the action, until the result is completely visible in the browser. An action might be that the user clicks a button, follows a hyperlink, presses the enter key in a search field etc.

To analyze and compare performance and scalability properties for Architecture 1.0 and Architecture 2.0, a number of system tests have been designed and executed in real deployment scenarios. Equal tests have been run on both prototypes. Now, one important thing to mention is that the performance of the architectures to some extent depend on the Web framework.¹ So considering the fact that the architectures use two different frameworks (Node.js and SpringMVC), might cause subjective results for comparison. However, I was aware of this fact before I chose to use two different Web frameworks, and therefore I built Architecture 2.0 with SpringMVC as well (not really a big effort, just a matter of implementing the API in Java). The performance results for Architecture 2.0 on Spring vs Node.js were very similar, and therefore I chose to do all the testing based on the Node.js version.

A final goal for this project is to analyze and compare the source code quality for the two architectures. The source code is to be measured in terms of code flexibility and maintainability, which is an important property in order to facilitate future code modifications and extensions. Now, such a comparison is a difficult and laborious task, and it is somewhat difficult to measure code quality. Therefore the analysis of the source codes

¹The interested reader can check out[45] for an extensive performance comparison of popular Web frameworks.

Operation System	Fedora 12
CPU Architecture	Intel(R) Core(TM)2 Quad CPU Q8400
CPU Clock Speed	2.66GHz
CPU Cache Size	2048 KB

Table 7.1: Computer specifications for the test server

is somewhat short, and partly given in terms of opinion. However, a proper test case has been designed and implemented on both architectures, and the results are relevant. We look at this in the end of the chapter.

In this chapter we will look at the tests that have been designed, and the results for these.

7.2 Web Application Performance

The response time for Web apps depend on many different factors. In general, these are:

- Application server's throughput, which concerns how many requests the application server can handle per time unit.
- Database server's throughput, which concerns how many transactions the database can handle per time unit
- Client-tier efficiency which concerns how fast the browser can render and display the result of a user-action. Often this depends on the JavaScript implementation that is required to display the result.

There are also other factors that affect the response time of a Web app, such as network performance and the hardware that hosts the client and the server. However, these issues will not be considered in this thesis, mostly because performance tuning these elements does not directly indicate any pros or cons in the two Web architectures that have been studied in this thesis.

Another performance concern is the application server's scalability. Scalability is a measurement of resilience under ever-increasing load. As such, another goal is to maintain the performance levels when the number of concurrent users increases, and to support as many simultaneous users as possible. Scalability also depend on the application server's throughput, and the database server's throughput.

7.3 Hardware and Software Used for Testing

Both prototypes have been deployed on a test machine stationed in Madrid, Spain. The machine was hosted by PlanetLab[98], a global network of computers made available for researchers to develop, deploy and test distributed systems. The reason I chose to deploy it in Spain was in order

to get realistic transmission times. The system specifications for the test machine are given in table 7.1.

Performance testing was made with the Chrome Developer Tools[25], which is a browser feature for the Chrome browser[24] that captures URL requests, and monitors JavaScript executions. The tool not only calculates the complete response time, but also gives a detailed overview of the times spent for each individual Web resource that is fetched from the server. Both Mozilla Firefox and Internet Explorer have similar tools for Web performance testing and profiling, and they mostly deliver the same functionality. I chose to use Chrome simply because I am familiar with it, and it doesn't require any extra plugin installation. Stress testing the server was done with Apache JMeter[9], which is a Java program that is able to execute and monitor multiple threads. These threads can be configured to do HTTP requests.

7.4 Performance and Scalability Tests

In this section, we look at the concrete tests that have been made, and the results of these. The tests are separated in five different sections. The first two tests investigate the performance of the prototypes by inspecting request response times, the third test investigates the scalability characteristics of the prototypes by stress testing the back-end implementations, the fourth investigates database speed, and the last test investigates the architectures' source code in terms of flexibility and maintainability. This analysis lacks a test case that investigate the database's throughput properties. Unfortunately I did not have time to create this test.

For testing purposes, an equal set of dummy objects was created in the two databases. These are:

- 1000 Shredders
- For every Shredder, a number from 0 til 10 fanees
- 100000 Shreds
- 100000 Battles
- 10000 Battle requests

7.4.1 Test 1 - Page Loading Tests

Goal: To determine how fast the prototypes create the major pages of Shredhub. Also, to investigate the times spent in the various phases during a request

Tests: Response time

The page loading test is meant to investigate the amount of time the user has to wait from the time a URL is requested, until the page is displayed in the browser. This test is important in order to identify which of the two prototypes are capable of creating a page the fastest.

The page loading test was performed actively by a human user. The tester would either, for the initial page, write the URL in the address bar of a Web browser and press the enter button, otherwise, the tester would click on a link inside Shredhub that leads to the given page. I have chosen to use the Chrome Web browser to run the tests, because it comes with the Chrome developer tools. Every test was done 5 times, in which the test results show the average of these.

The result figures show the complete round trip times for all the different server requests. These results are displayed in waterfall figures, which naturally depict the ordering of the different resource fetches. The waterfall models also capture the fact that some resources depend on each other in order to start fetching. For example, the browser will start fetch CSS/JS resources as soon as it has gotten the HTML page from the server. Every result figure shows:

- Time taken for the server request
- Time taken fetching CSS and/or JavaScript resources
- Time taken fetching images
- Time when the DOMContentLoaded, and OnLoad events are triggered
- Total time spent on server
- Amount of data received from the server (without images)
- Page rendered: the time it was fair to say the page is visible in the browser

. The results are measured in milliseconds. The last timing represents the time when everything but the images were completely rendered. This is because rendering images is the last thing the browser does (in the case for Shredhub), and it says nothing about the performance differences for the two prototypes. For Architecture 2.0, I have chosen to indicate the amount of time spent doing AJAX requests to the server (the timings concern the time it took on the server + HTTP transmission times), as well as showing the time of the **last JavaScript execution** in the *App*. The last JavaScript execution indicates the time when the *App* has rendered every necessary HTML template and written this to the Dom. Thus this is when I acknowledge that the page is finished and displayed in the browser. As for Architecture 1.0, the first request in the waterfall model always represent the time it took to process the request on the server plus the HTTP transmission times regarding this. I acknowledge that the page is finished and displayed in the browser when the DomContentLoaded event is triggered, because this is when the browser has rendered every element in the HTML page.

Note that the tests were performed on a somewhat slow network connection. This explains the slow timings, especially for image resources.

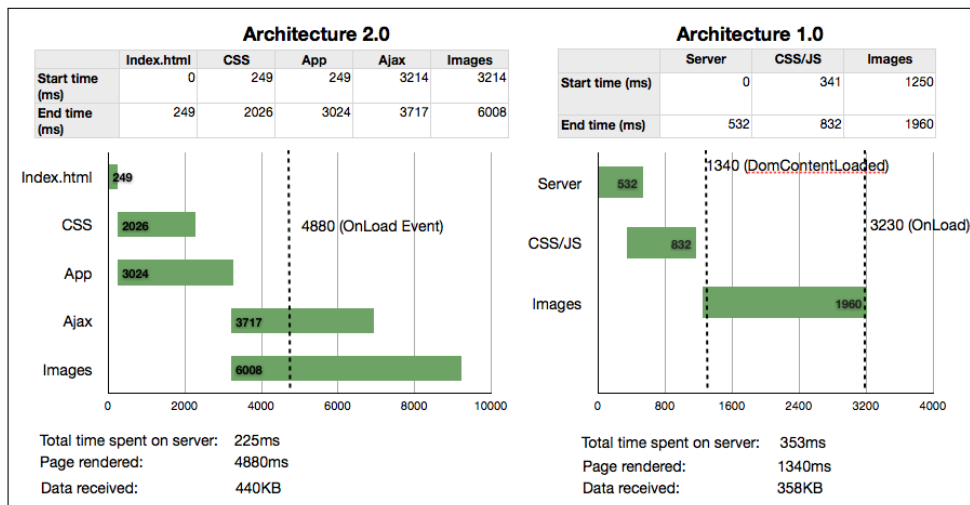


Figure 7.1: Test results for www.shredhub.com

However, because the testing on the two prototypes was done on the same network connection, the network overhead isn't relevant for the test results.

The URLs that have been tested are:

1. *www.shredhub.com/*
2. *www.shredhub.com/theshredpool*
3. *www.shredhub.com/shredders*
4. *www.shredhub.com/shredder/<UID>*

All of the pages except the first one requires the user to be logged in. In this case I have actively logged the user in before the real page loading tests were made.

Figure 7.1 shows the result for loading www.shredhub.com. Architecture 2.0 is very much slower than Architecture 1.0. The reason is that Architecture 2.0 has to load a big pile of JavaScript (the whole *App*!) before the browser can start to execute the JavaScript code in the *App* that build the page. The *App* also has to do an AJAX request to fetch the list of top shreds. Architecture 1.0 spends more time processing on the server, and the user has to wait 532 milliseconds before he can see anything at all on the screen. An advantage with Architecture 2.0 here, is that the user can see a minor part of the page already after 249 milliseconds. But this is just the scaffolding HTML that is contained in Index.HTML, which is basically just a "Shredhub" headline. However, it does give the user something to look at much quicker than for in Architecture 1.0. A final notice is that Architecture 2.0 sends more data to the browser. This is primarily because of the size of the *App*.

Figure 7.2 on the following page shows the result for loading www.shredhub.com/shredpool. Architecture 2.0 is displayed a little bit faster in the browser than Architecture 1.0, however the reason it is somewhat slow

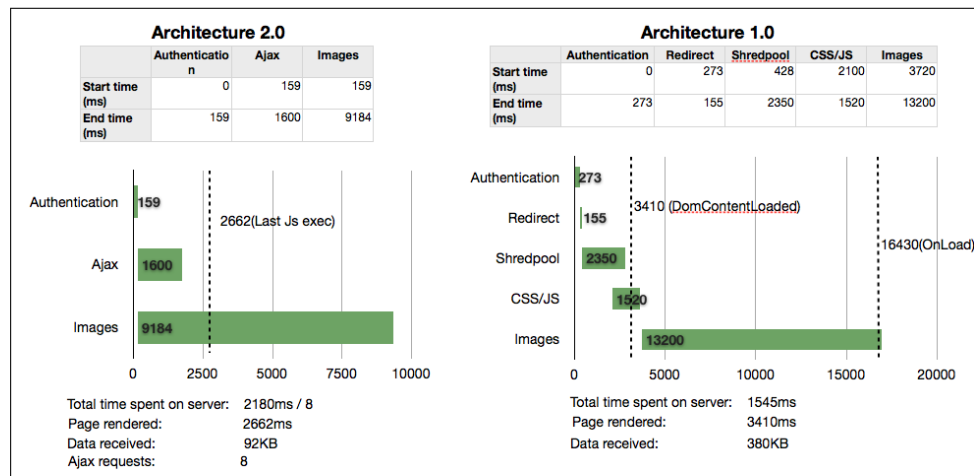


Figure 7.2: Test results for www.shredhub.com/shredpool

is because it has to perform 8 AJAX requests. The browser executes these in parallel (hence the 2180/8 ms for time spent on server), but it is still very time consuming. Architecture 1.0 is slow because it has to perform an HTTP redirect after authenticating the user. The real work happens during the work on rendering the Shredpool on the server. Altogether, Architecture 1.0 spends less time on the server, but is slower because of the redirect. Also, Architecture 1.0 sends more data. This is because the Shredpool HTML file is quite big.

Figure 7.3 on the next page shows the result for loading www.shredhub.com/shredders. Architecture 2.0 is almost twice as fast as Architecture 1.0. This is because Architecture 2.0 only fetches a small set of JSON Shredders from the server, and executes only a little bit of JavaScript in order to render the new page in the browser. As for Architecture 1.0, even though the execution on the server is quite fast, the result shows up late in the browser because the page that is sent is quite big and it is time consuming for the browser to render the whole page. Again, Architecture 2.0 sends much less data to the browser than Architecture 1.0. It's just an array of 20 JSON Shredders.

Figure 7.4 on the facing page shows the result for loading www.shredhub.com/shredder/<UID>. Again, Architecture 2.0 scores better, reasons being mostly the same as in the previous test. The App merely has to perform a tiny bit of JavaScript in order to create the new page, and there is just one AJAX call to the server in order to fetch the necessary data. Architecture 1.0 on the other hand is slow for the same reasons as in the previous example; the page that is created is big, and the browser has to render it from the ground up once it is received from the server.

7.4.2 Test 2 - Interactive User-Action Tests

Goal: Determine the response time for interactive user-actions on Shredhub

Tests: Response time

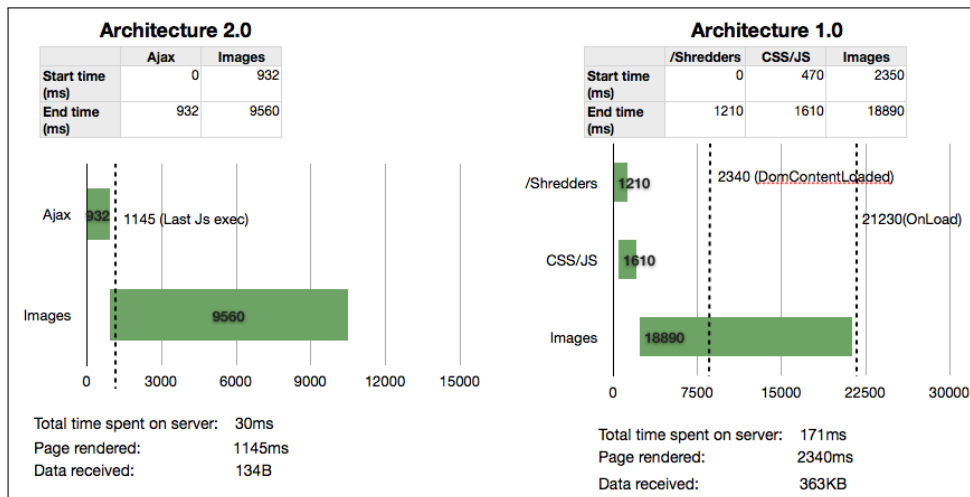


Figure 7.3: Test results for www.shredhub.com/shredders

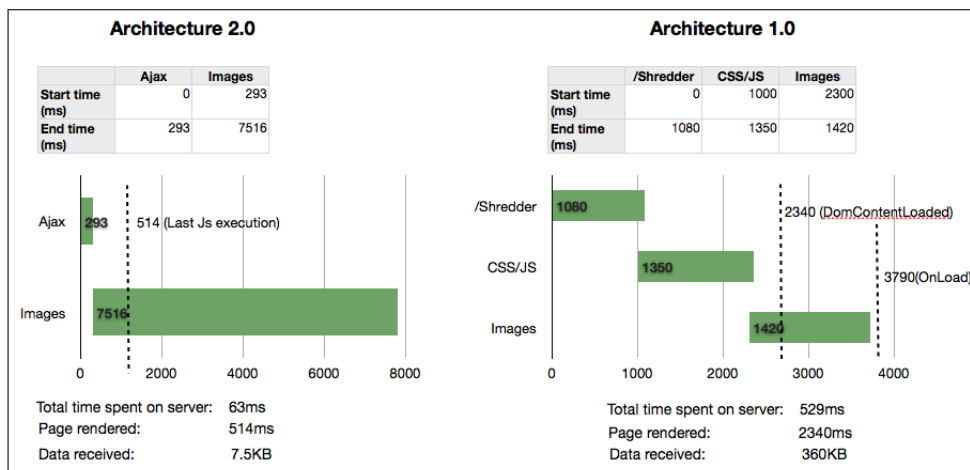


Figure 7.4: Test results for www.shredhub.com/shredder<uid>

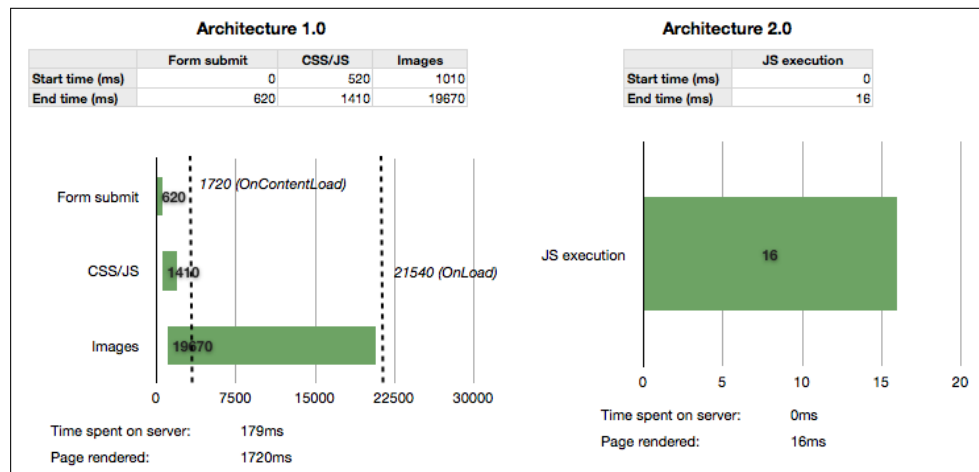


Figure 7.5: Test results for when the user clicks on next shred row

This test is investigating the response time spent when a user performs a particular interactive task on Shredhub. Unlike test 1, which is investigating response times when complete Web pages are requested, this test only concerns minor interactive actions that happens inside a page. Again, the results are showed in waterfall models that capture the various phases for each request.

The user-actions tested are:

1. The user clicks next on a shred row
2. The user clicks on a Shred that opens a new video window
3. The user comments a shred
4. The user rates a shred

Just like in Test 1, the Chrome development tool was used to investigate the round-trip time for each action.

Figure 7.5 shows the result for user-action 1. Architecture 2.0 is very much faster in this case. The reason is that it doesn't have to consult the server; the next set of Shreds was fetched when the Shredpool was first accessed, so it lives in the browser's JavaScript heap memory. It only has to render a few HTML templates and write the result to the DOM in order to show the new row of shreds. Architecture 1.0 is slower because it has to make an HTTP request to the server. Even though the next set of Shreds are cached in the session object, the HTTP round-trip adds to the cost of displaying the page. Notice also that Architecture 1.0 does a page refresh here, because it is an HTML form submit. This explains why so much data is sent from the server; Architecture 1.0 has to prepare the whole page on the server and send it to the client.

Figure 7.6 on the facing page shows the result for user-action 2. The results are basically the same as in the previous test. The only difference is

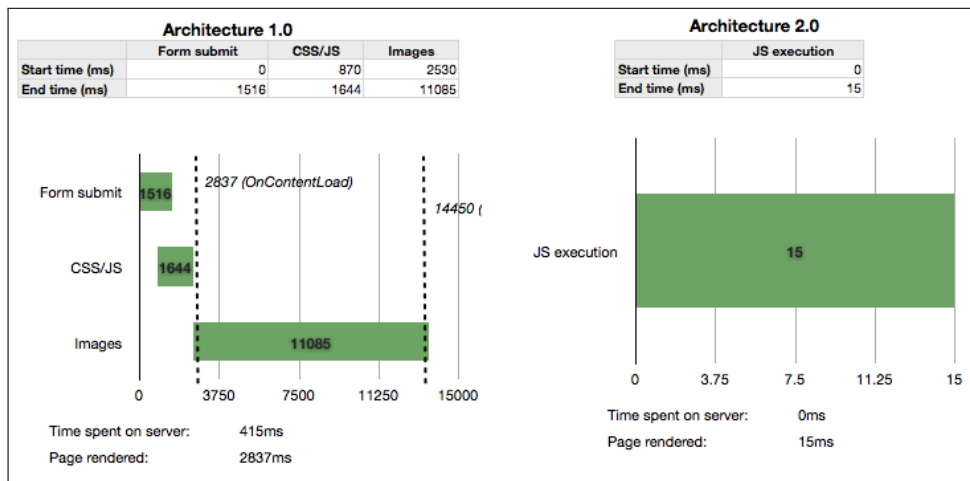


Figure 7.6: Test results for when the user opens a Shred window

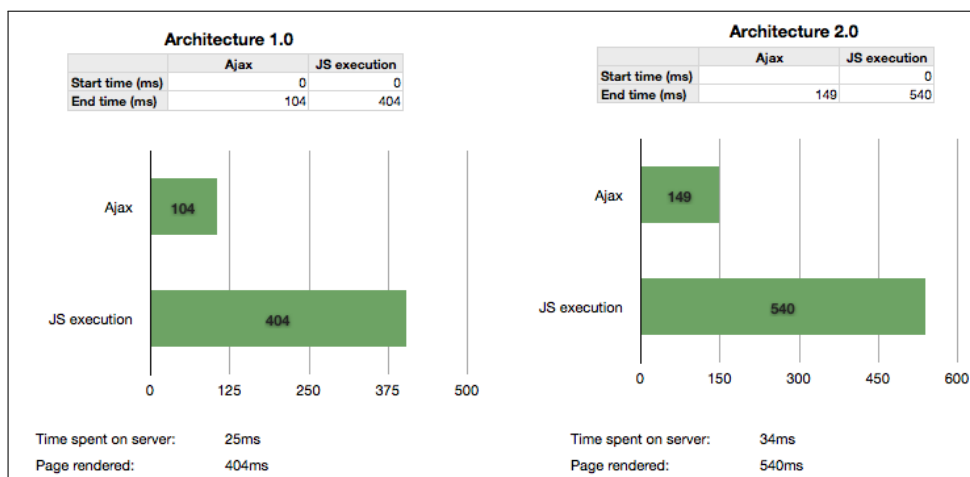


Figure 7.7: Test results for when a user comments a Shred

now Architecture 1.0 spends even more time on the server, because it has to query the database for the Shred.

Figure 7.7 shows the result for user-action 3. In this case the results are fairly equal. However Architecture 1.0 is a tiny bit faster, simply because it doesn't execute as many JavaScript statements as Architecture 2.0. The reason for this is that the JavaScript in Architecture 1.0 are simple self-contained handler functions of less than 10 lines of code. The same functionality in Architecture 2.0 is implemented as part of a bigger code base, and has to go through several function calls and object instantiations in order to execute the action.

Figure 7.8 on the following page shows the result for user-action 4. The results here are fairly equal to the previous test. Architecture 1.0 scores a tiny bit better because there is a lot less JavaScript to execute in order to

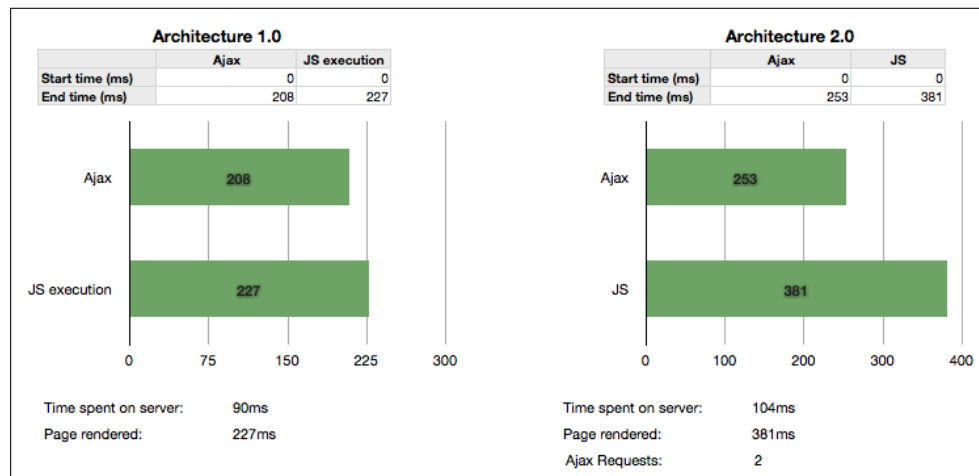


Figure 7.8: Test results for when a user rates a Shred

handle the request.

7.4.3 Test 3 - Back-end Scalability Test

Goal: Determine how many concurrent requests the prototypes can support, and how quick the server handles requests under heavy load.

Tests: Scalability of the back-end implementations

This test was performed by creating multiple threads that executes a set of predefined actions on Shredhub. The actions are meant to simulate a normal flow of user-actions, to get a best-as-possible view of how well the server scales under common user-scenarios. The tests were created and executed with Apache JMeter. This was configured to have one test case that issues many subsequent actions:

1. The user visits the home page
2. The user logs in and visits the Shredpool
3. The user uploads a shred
4. The user watches a shred
5. The user comments a shred
6. The user accesses the page www.shredhub.com/shredders
7. The user clicks on a particular shredder, which leads to the page www.shredhub.com/shredders/<uid>

Each thread (that is, a user) executes all these actions on the server, once. An appropriate “thinking” time was also added between every action.

To perform stress testing, JMeter was set up to generate an increasing amount of simultaneous threads until the server starts to return erroneous

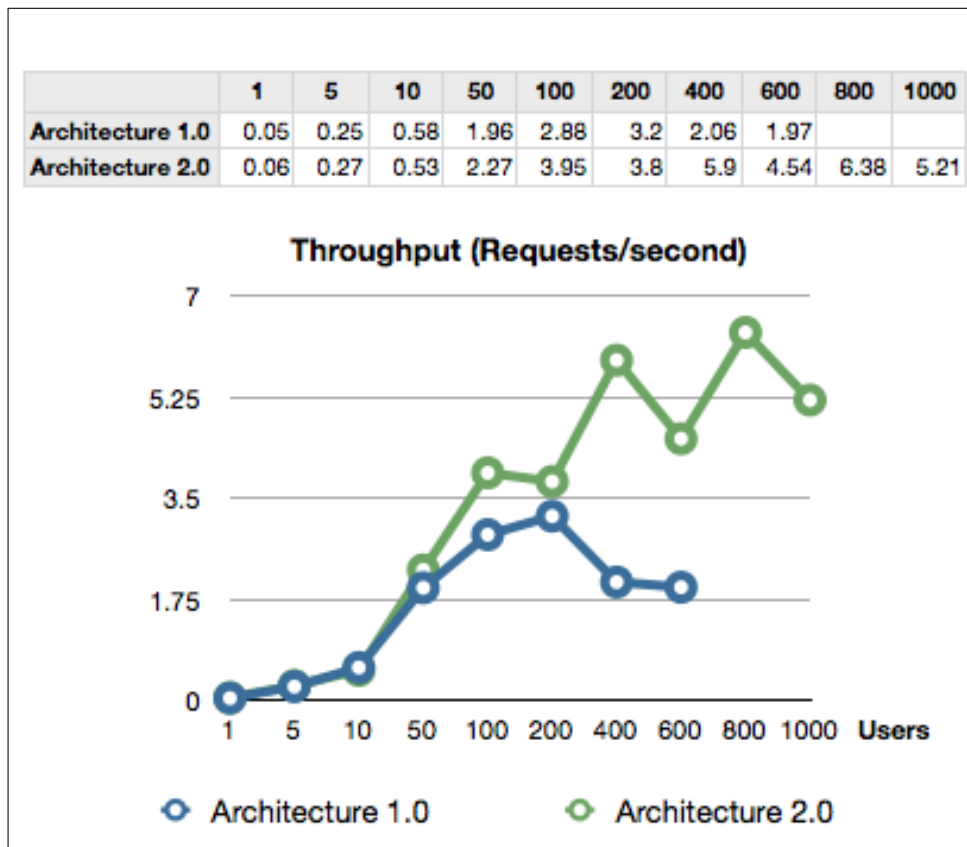


Figure 7.9: Average number of requests handled per second

responds. Now, JMeter needs a given amount of time in order to be able to create enough threads without saturating the test computer. This is called the ramp-up time. JMeter was configured to create the number of threads T with a ramp-up time = N seconds, where $N = T$ for values of T from 1 to 100. All subsequent amount of threads T were created with a ramp-up time of 100 seconds. The ramp-up period was configured this way, also in order not to create an unusual high hit rate on the server, which would be an undesirable condition. The client test machines weren't able to issue more than 300 simultaneous threads executing the test case. Therefore, I had to add a new test machine for every $n \cdot 300$ threads. The results are showed in figures 7.9 and 7.10 on the following page.

The results show that Architecture 2.0 can handle more concurrent users than Architecture 1.0. In addition, Architecture 2.0 has relatively quick request times compared to Architecture 1.0. This makes sense, considering the server spends a lot of time rendering the HTML page and doing session and state management. A misfortune with Architecture 1.0 is that after 100 users, the request times were very high. The reason for this is mainly because of the high memory consumption on the server when the number of active users is high. The server can handle them, but the processing time is slow. I chose to stop increasing users after respectively

	1	5	10	50	100	200	400	600	800	1000
Architecture 1.0	3.7	4.1	2.7	2.7	3.5	15.3	59.5	99.8		
Architecture 2.0	0.6	0.4	0.43	1.45	2.5	3.3	9.7	15.1	16.3	17.1

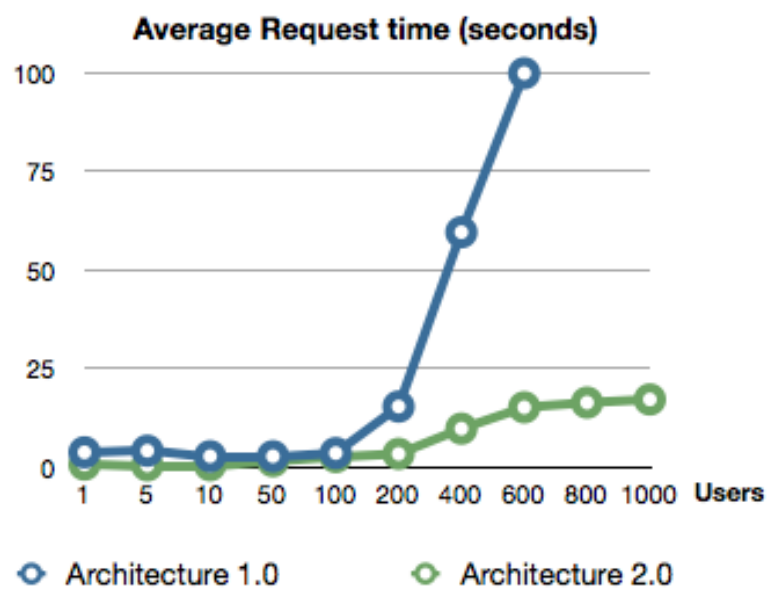


Figure 7.10: Average time spent per request

Query	CRUD	Architecture 1.0	Architecture 2.0
ShredsByRating	R	244ms	241ms
ShredsByFanees	R	11ms	68ms
ShredsByFaneesOfFanees	R	202ms	497ms
ShredsByTags	R	212ms	56ms
CreateShred	C	64ms	30ms
AddShredRating	U	27ms	16ms
DeleteShredComment	D	33ms	11ms

Table 7.2: CRUD operations tested

600 and 1000, because after this, the error percentage increased drastically.

7.4.4 Test 4 - Database Performance

Goal: Determine how fast the most common database queries in Shredhub executes

Tests: Database efficiency

This test investigates the speed differences in using a MongoDB (NoSQL) database versus a SQL database for Shredhub. The test was done by timing the query times spent when a user performs a given action that requires a database operation. A set of the most commonly used database queries were used as test cases. The timings start from the time the query instruction is made by the calling database handler, and ends when the result is mapped to a domain object (i.e a Java object for Architecture 1.0, and JSON object for Architecture 2.0). The test cases are designed to inspect at least one query from each of the four CRUD operations. Multiple Read operations have been checked, however, considering there are many different types of read operations in Shredhub. Every query was issued 10 times, with a different user, in which the results depict the average. The results are showed in table 7.2. The results are fairly equal, however, there are some differences. Operations for Architecture 2.0 that requires MongoDB to perform custom join operations, are generally slower. This concerns ShredsByFanees, and ShredsByFaneesOfFanees. Operations that require SQL to join while MongoDB does not have to join because of its nested structure results in performance gains for Architecture 2.0. This concerns ShredsByTags and create, update, and delete operations.

7.4.5 Test 5 - Code Flexibility Test

In this section we look at a test case that is designed to test the code flexibility and simplicity in the two prototypes. In this test, a new interactive user feature was to be implemented on Shredhub. The test is designed in a way that involves the modification of the user interface, implementing a new business process, and alteration of the database. The results for this test outlines the number of code statements that were added and modified, and the programming language(s) that were used to implement the feature. Also, because code quality is somewhat difficult to

measure, some significant properties are pointed out for each result. The new user feature that was implemented is given below.

User feature: The Guitar Showroom The Guitar showroom is part of the user's profile page, where the user can have pictures of his guitars, and other users can view the images one-by-one by dragging them to the left or right. The scrolling must be highly interactive, meaning no page refresh can happen. Also, for every picture, the user can choose to "dig" the guitar. By clicking "dig", the guitar earns a "dig" point. Dig points is a way for the user to show that he likes the guitar. Each time a user digs a guitar, the owner also earns one experience point. The user is not allowed to dig one of his own guitars.

For testing purposes, a set of the existing test Shredders are to add an image for one of their guitars, and a fictive dig-value for it is given (i.e changing some of their guitars to showroom guitars).

Result for Architecture 1.0

The code that was added and modified in order to implement the feature is given in the table below:

JSP + 47 lines (HTML + JSP script statements)

JavaScript + 81 lines.

Java + 73 lines, - 10 lines, + 1 class

SQL + 1 table, - 1 column + 1 column migration for every Shredder.guitars[]

In the front-end, a set of new specialized and tightly coupled JavaScript functions were added to the end of the Shredder.JSP file. The amount of such specialized and independent JavaScript functions that doesn't have any structure or reusable modules are starting to add up. If more such user-interactive behavior is to be added later, the result will possibly continue and add up in terms of tangled and inflexible, specialized JavaScript code. The back-end, however, did not require much modification, except for the database mapping code that was required in order to handle the change in the SQL structure.

In the Java codebase, a new controller handler, service function and DAO function was implemented. In addition, the ShredderMapper class had to be modified in order to handle the update in the SQL structure. Also, a new domain class had to be created to hold the new SQL update: **GuitarForShredder.java**. Now, every Shredder has an array of GuitarForShredder objects, as opposed to earlier, when every Shredder had a simple String array of guitar names.

In SQL, a new table was generated to hold an image string, digs int, and name string, and a reference to the Shredder who owns the guitar. Previously every Shredder table had a simple array of strings that represented the guitars they own. Now, in order to implement the

new update in SQL, I first had to create the new table GuitarForShredder, then I had to move the data from each shredder.guitars column into a new guitarForShredder row. Finally the old shredder.guitars column was deleted. The alteration of a Shredder's guitars is given in the example below:

```
// Old Shredder table (only the guitars part):
Column | Type |
guitars | text[] |

// new guitarsForShredder table
Column | Type |
guitar | character varying(50) |
shredderid | integer |
imgpath | character varying(20) |
digs | integer |
```

Result for Architecture 2.0

The code that was added and modified in order to implement the feature is given in the table below:

HTML + 40 lines - 4 lines

JavaScript App + 88 lines.

JavaScript API + 19 lines

In the *App*, the new view logic code was added to an existing module. A couple of new event-handlers were added to the Shredder view, and a business operation was added to the Shredder model. Part of the business rules had to be duplicated to the back-end, to avoid illegal misuse.

On the back-end, there was no need to alter the database, because of MongoDB's flexibility: Previously, every Shredder had an array of guitars as strings that represent their list of guitars. Now, for every guitar that is to include an image and a dig int, the array index that used to represent that particular guitar could simply be altered to be a nested JSON object inside the array, instead of a simple string. This would only have to be altered for those test Shredders that were to change their old guitar from being just a name, into a JSON object with name, image and dig value. There was no need to alter any of the other Shredders, or even alter any of the CRUD operations that touches the Shredder object. The alteration of a Shredder can be seen in the code below:

```
// Old guitar array:
Shredder {
  guitars : ['Gibson Les paul', 'Fender Stratocaster']
}

// Showroom guitar
Shredder {
  guitars: ['Gibson Les Paul',
    { name : 'Fender Stratocaster',
      image : 'fenderStrat.jpg',
```

```
    diggs: 34
  }
}
```

The only modification needed at all was actually in the HTML template that uses the list of guitars: an if-else block had to be added in order to check if a guitar is a string or a JSON object. Hence the -4 HTML lines in the listing above.

Also, the back-end had to be given a new REST api function to handle the database update for adding a dig.

The code that was implemented for these two tests can be seen at the end of Appendix A and B.

7.5 Summary

In this chapter we have looked at the tests that were made in order to analyze and compare the two prototypes. Five different tests were created that analyze performance, scalability and code flexibility/simplicity.

7.5.1 Performance Results

For the page creation tests, Architecture 2.0 was generally faster because it only has to create the part of the HTML that was meant to change, which is done quickly inside the browser. Architecture 1.0 has to create the whole page in every request, which also entails a complete browser rendering process (page reload).

In the interactive user-action tests, Architecture 1.0 was slower in cases where it did HTML form-submits, because it led to the generation of a whole new page. In other cases, the results were fairly equal, but a bit better for Architecture 1.0 because it executes less JavaScript executions than Architecture 2.0.

In the database tests, the results were fairly equal, but the NoSQL solution is somewhat slow in queries that require manual join operations. In other cases it turned out faster than the SQL database.

7.5.2 Scalability Results

In the scalability tests, Architecture 2.0 came out the best, simply because the server does a lot less processing. Architecture 1.0's throughput was also significantly lower when the amount of users was high, and in addition, Architecture 2.0 managed to serve almost twice as many simultaneous users as Architecture 1.0.

7.5.3 Code Quality Results

The code quality test showed that the NoSQL solution was more flexible than SQL; because of its schema-less approach, no table alteration or data migration was needed when parts of the domain had to change.

The amount of code needed for Architecture 2.0 was also less than in Architecture 1.0, thus one could argue the codebase is somewhat simpler. Much of the reason is that Architecture 2.0 avoids the tedious marshalling of objects that are sent from client to server, and server to database. Also, adding more JavaScript to Architecture 1.0 resulted in an increase in the number of specialized and non-reusable functions. However, in Architecture 2.0, some business rules had to be duplicated on the back-end.

Part III

Discussion and Conclusion

Chapter 8

Discussion

8.1 Introduction

In this chapter we discuss the results that were found in the previous chapter. We discuss the problem statement in respect to the results, and propose possible hybrid solutions that combine the strengths of the two architectures. At the end we discuss the strengths and limitations of the study, and how the results should have an impact on future practice and research.

8.2 Page Rendering

Test 1 and 2 clearly showed that rendering on the client gave faster response times for the user. The reason is that the browser maintains all the HTML that is ever needed for the whole *App*. Therefore it does not have to ask the server for HTML when the user goes to a new page or performs an action that requires new HTML to be rendered. It will simply just be fetched from the browser memory, and merged together with the necessary JSON data. The client still has to fetch the JSON data from the server from time to time, before it can do all the rendering, but this is not a matter of large data quantities. What is even better, is that the JSON data is fetched asynchronously. This way, when the user goes to a new page, the JavaScript code will immediately show the new page, except for just the content that requires JSON data. When the server request returns, the JavaScript *App* will finish the page by adding the JSON content to it. Thus the user sees part of the page request almost immediately, in which the rest comes reasonably quick after. Good examples are figure 7.3 on page 93 and 7.4 on page 93, where the whole page is completed as soon as the browser receives JSON data from the server (even though part of the page is visible even earlier).

Also, when the client performs the rendering, it can choose to render only the parts that are necessary in order to display the result of a user action. In the case for server-side rendering, it has to render the whole page in any case. In addition, this leads to browser page refreshes, which is an unfortunate user experience.

Rendering on the client also limits the amount of bandwidth consumed, which is proved in Test 1 in the previous chapter. Looking at the numbers regarding the amount of data sent between the client and server, shows that the JSON-based solution in Architecture 2.0 consumes much less bandwidth. The reason is that the fine-grained JSON solution consumes far less data than the HTML-based solution from Architecture 1.0.

An important decision is whether to eagerly fetch all the HTML and JavaScript at first, or to fetch it (lazily) when needed. I argued that eager fetching was preferable for Shredhub, because when all the HTML templates were merged together and compressed, the complete size was small enough to send in the initial page load without it being too much. However, this has not been tested on clients with poor browser capabilities and/or poor networking hardware, and it might be that lazy fetching might be a preferable solution for some clients. Also, if Shredhub is to grow extensively in size with new pages and user features, it will probably be desirable to fetch the resources lazily regardless.

A disadvantage with server-side rendering is that it is a time consuming process, something that is clear from figures 7.3 on page 93 and 7.4 on page 93 where a lot of time is spent on the server. An outcome of this is that Architecture 2.0 can handle many more simultaneous users, because the average request times are much higher (table 7.2 on page 99). Thus, the architecture is a lot more scalable.

Now, figure 7.1 on page 91 shows that rendering on the server results in faster response time for the initial page request. The reason for this is that in Architecture 2.0, the browser has to wait for the whole *App* to be completely loaded in the browser before it can start fetching additional JSON data and render the page. This is very time consuming and the tradeoff can in some cases be too high, depending on where one sets the upper limit for initial page response times. A proposal for a hybrid solution in this case is where the home page is rendered on server and includes a reference to the JavaScript *App* with a “*async*” tag such that the *App* is fetched lazily after the page is rendered in the browser. The result might be that the home page is rendered faster and the application will still have the thick-client solution for the rest of the user-interactions.

8.3 State and Business Logic on the Client

Moving application state and business logic to the client has clear performance advantages for Shredhub. The browser stores state-data in its JavaScript memory and local storage, which in many cases avoids the need to consume the server for data. Good examples that proves this are in figures 7.5 on page 94 and 7.6 on page 95, in where the browser doesn’t even have to fetch data from the server. When the client does have to fetch data, it consumes the API, which is always done asynchronously with AJAX, and in effect doesn’t lead to any browser page refreshes. The outcome is a much faster Web app, with a highly interactive user experience where the user almost doesn’t notice the HTTP requests. This naturally leads to better

scalability on the server, as less time is spent there.

One disadvantage with this, however, is that it requires a lot of JavaScript code in order to keep the code base maintainable, which might lead to a lot of source code that is sent to the server. Much of the reason for this is that the JavaScript language lacks some important language features, and there are still browser incompatibilities that require a lot of extra JavaScript frameworks to be downloaded to the browser. Figures 7.7 on page 95 and 7.8 on page 96 showed the advantages of the simple JavaScript handlers in Architecture 1.0, which led to slightly better response times. However, the tradeoff is that these simple JavaScript handlers do not facilitate code reuse or any modularity, because they are implemented as simple handler functions that merely does one specific task. In the long term, this might add up to tangled and messy code. Disadvantages with this might be that the code can be difficult to read, variables can be overridden due to scope issues, refactoring the code can become tedious, and the code is difficult to test. A clear indication of this was found in test 5 in the previous chapter, where 81 lines of JavaScript code was separated in a few tightly coupled JavaScript functions, in order to build a new interactive user feature for Shredhub. Architecture 2.0, on the other hand has a more intuitive and coherent code base, because both the view logic and business logic is implemented in the same language. This way it is possible to gather view logic and business logic that concerns the same domain under the one module, and thus achieve more coherence, while at the same time keeping HTML templates highly clean and almost stripped from template script tags and JavaScript. Architecture 1.0 uses three different languages for view and business logic (JSP, JavaScript and Java), and much of the view logic is tightly coupled with the HTML. Thus the programming benefits for the view logic code are clearly superior for Architecture 2.0.

Another advantage with Architecture 2.0 is that the use of the back-end API decouples the client from the server. In Architecture 1.0, each HTML form and hyperlink has an associated controller handler on the server, that returns a particular view. The API however, is more general in that it doesn't return a view, because it is up to the client caller to decide how to use the results. Also, most API calls are flexible in that they allow the caller to for example define result sizes and page numbers. Now, the big advantage with this is that it allows other client users to use the API as well. This could be a future mobile app, or a 3rd party application that wishes to use Shredhub data. However, due to the three-layered architecture in Architecture 1.0, offering a similar API would simply be to build a new presentation layer on top of the domain logic layer, that supports RESTful operations and communicates a more fine-grained data format, such as JSON or XML.

Now in addition, avoiding sessions on the server had a very high performance impact. In Architecture 1.0, the server had to maintain a large set of Java objects in memory, for every current user. This resulted in slow response times when the number of simultaneous users was high, and the server was not able to handle more then 600 simultaneous active users (see table 7.2 on page 99).

Now, it is difficult to state how much of this limitation was caused by the amount of memory consumed for maintaining state and sessions on the server, and how much was due to the complex rendering processes that happens for every request. I acknowledge the fact that this thesis lacks a deeper investigation of this in order to be able to draw more concrete conclusions regarding the scalability issue for Architecture 1.0. A solution would be to use a profiling tool to inspect how much time was spent on state handling, versus template rendering, and to use a monitoring tool to verify the amount of memory consumption used in maintaining session objects.

Another hybrid solution in this case would be to use Redis for Architecture 1.0 to store sessions and state data. This would limit the memory usage for the Java virtual machine that currently maintains this data. The success with using Redis for authentication in Architecture 2.0, might be transferable to Architecture 1.0, because the use case is very similar.

An important lesson learned from the performance test is that HTML form submits primarily used to implement minor interactive behavior (like in figure 7.5 on page 94 and 7.6 on page 95), should not result in complete HTML pages, but a more fine-grained server response like the ones in Architecture 2.0. A lot of unnecessary page generation is done just for a small DOM alteration.

Architecture 1.0 can also benefit from a similar JavaScript code structure like the one in Architecture 2.0, in order to avoid having the JavaScript add up to a pile of tangled and non-reusable functions.

A disadvantage with having the server completely stateless becomes clear in figure 7.2 on page 92, in which case a lot of time is spent on the server fetching JSON from the database. Now, this is done over 8 different HTTP requests, which results in a high transmission overhead and slow response time. Even though the complete rendering process is faster for Architecture 2.0 from the user's perspective, a lot of time could have been saved if the server was aware of the 8 different database fetches that are required to display the Shredpool. Another solution would be to offer a more coarse-grained API function that simply fetches all the JSON data that is needed in order to build the Shredpool page on the client, given a Uid for the Shredder. The tradeoff here is that offering such coarse-grained API functions could create tighter couplings between the client and the API, similar to Architecture 1.0.

Although the study shows many advantages for this thick-client architecture, there are some major pitfalls:

- Some business rules has to be duplicated on the server in order to prevent malformed user input. This could come intentionally from users who knows how to issue HTTP requests without using Shredhub.com's Web interface.
- The Web app might not perform as well on other client machines and browsers than the ones that was used for testing. This might be a serious pitfall, because the result might be that slow

computers and/or old browsers execute the JavaScript code so slow that Architecture 1.0 might be a preferable solution in terms of performance. This is most likely a case for older smart phones and desktop computers. I also acknowledge the fact that the testing phase of this thesis should have been done more extensively on various computers and smart phones in order to support these statements. Unfortunately, I did not have the time to do this.

- The architecture is also not optimal for search crawlers, in where crawlers inspecting Shredhub would find merely empty HTML tags without content. Now, this is not a very critical problem, because most of Shredhub's content is only to be viewable once logged in. However, some parts of Shredhub should be fully searchable on the Web, and therefore a better solution for this problem remains to be implemented.
- Some services depend on the number of times a page is reloaded. Examples are advertisement services like Google AdSense[52], and monitoring services like Google Analytics[53]. With architecture 2.0, these services wouldn't get correct page load data, because the page only loads once, on the initial request.

8.4 NoSQL and SQL Implementation

The database test for Shredhub was somewhat limited, however the results show some valuable points that are worth discussing. First of all, the MongoDB implementation does not perform particularly fast when manual join operations have to be done. This concerned the read operation get Shreds by fanees, in which the JavaScript caller first has to fetch the Shredder, extract his array of fanees, then do a fetch operation for all Shreds where the owner is in the set of fanees. Also, an even worse case is when it has to fetch Shreds made by fanees of a Shredder's fanees. Here, the JavaScript caller has to further fetch all the Shredder's fanees' fanees and search for Shreds where one of them is the owner. Now, I should have added an index for the owner of a Shred in order to speed up this execution. In PostgreSQL, the Shred owner is a foreign key, in which the database has already added an index for it.

Even higher speed results could have been achieved if I had chosen to add indexes for quicker sorting; for example, I should have added an index for the time a Shred was created, considering this is used as sorting key for most Shred and Shredder queries. This applies to both the SQL and MongoDB implementation.

On the other hand, MongoDB has clear speed advantages in cases where join operations are avoided. This happens because the MongoDB implementation wraps many of the separated SQL tables from Architecture 1.0 into one big collection, and therefore avoids having to join multiple tables together. The results show that create, update and delete operations are generally faster on Shredhub, because they are all just operations on

a single MongoDB document, as opposed to the separated tables in SQL. Even the range query “get Shred by tags” are faster, because tags is a nested string array inside every Shred. In Architecture 1.0, tags is a separate table in which case joins has to be done.

Also, Architecture 2.0 has a big programming satisfactory advantage; everything is written in JavaScript. This facilitates better and cleaner cooperation across the whole codebase. This is especially beneficial for the database wrapper, because the data structure used is JSON based, which makes possible to completely avoid database mappers. In Architecture 1.0, database mapper code makes up the majority of the back-end codebase. Now, because JSON is the data structure used as transmission medium in the API as well, no marshalling is needed here either. This way, the architecture saves a lot of source code lines. The point is proven in the number of lines of code in Architecture 2.0 versus Architecture 1.0: **10505 vs 15867**.

Another big advantage with the MongoDB implementation, is that the domain can be persisted exactly as it appears in the user interface. This is because of MongoDB’s schema-less approach, which makes the application more flexible. This came clear in the last test of the previous chapter. Here, a lot of manipulation had to be done on the SQL database in order to implement a new feature. In MongoDB, manipulation was completely avoided, because the database allows for collections of similar types to have different content (hence, schema-less).

A final observation is that using Redis made it possible to have a session-free server, and still authenticate users for every API request (except requests for the login page), without getting performance bottlenecks. A great advantage with this is that it facilitates a shared-nothing architecture, which again facilitates replicating the back-end to many server machines. This could lead to large performance and scalability gains. This is somewhat limited in the session-oriented architecture, because sessions do not apply so well in distributed deployments: If a session is created on one server, and the user is directed to another server in a later request, the session is not found, and the user will be directed to the login page. Also, if a server goes down, every session on that machine is lost. Now, there are solutions for these problems, but they still provide more distribution obstacles than Architecture 2.0 does.

8.5 Strengths and Limitations of the Study

8.5.1 Strengths

The thesis reveals many important aspects of modern Web architecture design. I believe the results are not specific to Shredhub, but demonstrate principles that are valuable for the traditional Web 2.0 applications discussed in the initial chapters. The thesis especially proves one important point: Moving demanding tasks to the client with JavaScript is not only feasible, but leads to increased scalability and improved response-times.

This trend is still very newfangled, and lacks proven results. I also believe that the thesis reveals pros and cons for both architectures, and is not biased by my own experience.

8.5.2 Limitations

One limitation with this study is that I do try to solve many different problems in one single project. One could argue that each problem statement does not have sufficient material to give unambiguous conclusions. I do state that the results from the tests are evident enough to answer the problem statement, however, I do acknowledge that I could have chosen to narrow the scope of the thesis. Parts of the reason why I studied all these different technologies was that I found it very educational. However, I should have tried to team up with another master's thesis student who could have done either the back-end or front-end part of the study.

Also, the code flexibility/simplicity test is a bit limited, and would possibly belong in a separate thesis by itself. However, even though it was difficult to measure, I do believe some valuable points were proven.

A final notice is that I have chosen not to focus particularly on various caching techniques, as this would be too much work for this thesis. Caching is absolutely necessary for high scalability and performance, which is why I mention it here.

8.6 Related Work

The thick-client Web architecture is not a particularly new concept. However, implementing them as pure large-scale JavaScript architectures is definitely a new approach, and therefore it is not a lot of research that benchmarks the performance benefits for this. Related work mostly concerns the use of AJAX technologies for improving modern Web applications.

In a study, Mesbah and Van Deursen[77], proposed a reverse-engineering technique for migrating a traditional Web application to a single-page application (a concept described in chapter 3). The study does not concern comparing the traditional approach with the single-page approach, it is merely a study of how one can move from one to the other.

Mazzetti *et al*[75] have implemented, in another study, the MIRAJ framework, a system built to test and validate how REST and AJAX architectures can be implemented to cooperate in providing a public Web API. The conclusion was that the architecture proposed was a valid solution for a modern Web application back-end.

In a study from 2009, Ohara *et al*[92] studied the data-centricity in a typical Web 2.0 application that incorporates heavy use of user-participation. The implementation has client-side presentation logic and AJAX for client-server communication. They concluded that the server uses much time in the data source layer doing database manipulation, because of frequent

AJAX requests. This could lead to reduced performance and poor scalability.

As for database technologies, some studies have been done that compares SQL with various NoSQL databases. Barahmand and Ghandeharizadeh[15] proposed a benchmarking system, BG, for databases. In the study, a BG benchmarking test for comparing performance of a MongoDB database, against a SQL database was made. Similar to this thesis, the study is based on a social networking application that is built with these two different databases. None of the solutions were superior, however when Memcached was added to the SQL solution, it had superior performance.

Cattell[19] did a comparison of many NoSQL and SQL systems designed to scale over multiple servers. The study merely concerns architectural comparisons, and does no benchmarking. The conclusion does not favor any of the databases, but argues for their respective pros and cons. The result of this study was important for this thesis, as the comparisons were used as a basis to decide the database solutions I have chosen.

8.7 Implications on Practice

I believe this study is important, because there are still many Web application developers who swear to many of the concepts of *Reference-model 1.0*. Also, many developers aren't aware of the possibilities to move the application to the client. In addition, many are not aware of the possibilities of the JavaScript programming language itself, for example that it is fully possible to build modular and flexible large-scale codebases with the language. This often leads to the type of JavaScript code that was implemented for Architecture 1.0, something I argue should be avoided. The thesis advocates the capabilities of modern browsers, which enable developers to build thick-client JavaScript Web apps. These thoughts are fairly new, and lacks research.

Even though the thick-client architecture doesn't necessarily fit every modern Web application, they are important concepts to contemplate. For example is rendering HTML on the client very relieving on the server, and leads to increased scalability and throughput. If a complete client-side rendering architecture is not an option, maybe choosing to perform some client-side rendering and some server-side rendering is possible. A possible hybrid solution could be to let the server be responsible for rendering the major pages, while the client renders smaller dynamic HTML changes inside the page. I leave this question as a task for further research: The study of finding solutions for combining server-side rendering with client-side rendering. This work would involve identifying when it is worthwhile rendering on the client and vice versa. This could also help improving the results picked up by search crawlers, something that is very important for applications that are not behind a login barrier.

In this thesis we saw pros and cons for both using MongoDB and SQL. Maybe, there are ways to combine these in the same Web application,

with the purpose of finding hybrid solutions. We saw for example that MongoDB could be used in combination with Redis in Architecture 2.0. This is an open question, and I leave others with an encouragement to further look for application areas where NoSQL solutions can be combined with SQL in order to fulfill each other. This work would involve finding parts of a domain where a given technology doesn't properly fit, in which case another technology might. This would require a very high decoupling of the data that is to be persisted, in order to separate it in different databases.

Finally, we saw that Architecture 2.0 could be built by using JavaScript both on the back-end and front-end. A problem, however, was that some business rules had to be implemented both on the back-end and the front-end. I did not try to look for a solution to merge these together with the intentions of avoiding code duplication. I do believe this could be possible considering it is only one overall programming language used, so therefore I leave this as a further research topic.

8.8 Summary

Performance and Scalability In this thesis I found that rendering HTML on the client takes much load off the server, and therefore makes it more scalable, because less processing has to be done for each request. This mostly resulted in better response times because the client could choose only to render the parts of the page that are necessary, and doesn't have to ask the server for the HTML. I also found that Web apps can perform better by having state and business logic in the client, because it endorses the storage of database objects in the browser's memory, and therefore reduces the amount of server calls needed. Once it must consult the server, it happens asynchronously in the background without the user noticing any delay, because there is no blocking process. This has great scalability benefits, also because the server doesn't have to maintain session data in memory for every current user. Also, the thick-client architecture decouples the client from the server, which might facilitate replicating the back-end. Which again is likely to improve the scalability of the system.

Finally I found that MongoDB might have performance benefits over traditional SQL, because its data structure can reduce the need to join documents together. However, it is much slower in cases where it has to perform joins. Also, using Redis was a good solution for the stateless server model, because it authenticates each request very quickly.

Programmer Benefits The programming benefits for the thick-client architecture is that both the user-interface logic and business logic is implemented in the same language, which creates a more coherent and intuitive code base. This also made it easier to have them cooperate together, as opposed to the thin-client solution where these responsibilities are separated in three different languages. Also, both of the NoSQL databases gives a very satisfactory programming environment, because

only one common programming language is used, and therefore avoids the need to implement data-marshalling. In addition, the NoSQL approach is very flexible and allows the domain to be persisted exactly as it appears in the user-interface.

Future Alternatives We also saw some possible hybrid solutions where principles from Architecture 2.0 can be applied in Architecture 1.0 and vice versa. The most alternative is to combine client-side and server-side page generation, in where the server generates the coarse grained pages, while the client is responsible for generating smaller parts of a page. Also, possible future studies involves investigating how SQL and NoSQL solutions can be combined in the same application in order to fulfill each other when the technologies themselves become a bottleneck. Another interesting study is in cases for pure JavaScript Web applications; investigating how source code can be shared between the front-end and back-end, in order to avoid the code duplication that occurs in Architecture 2.0.

Chapter 9

Conclusion

In this thesis we have investigated traditional and innovative architectural principles for modern Web applications. We defined a Web app as a traditional Web 2.0 application that includes highly interactive behavior, social networking features, and large quantities of persisted data. The problem statement concerns architectural alternatives for implementing such applications. It asks the question of whether the application can benefit from doing HTML rendering on the client, and if there are advantages for moving business logic and state handling to the client. The motivation for this is new improvement in JavaScript engines in modern browsers, which enables more complex JavaScript executions on the client. Also, the problem statement addresses modern database solutions called NoSQL, and asks whether there are any such database that suits the JavaScript-oriented Web application.

In order to solve this, we defined two reference-models; *Reference-model 1.0* and *Reference-model 2.0*. These address principles for a traditional thin-client Web application architecture, and a modern and innovative thick-client architecture, respectively. *Reference-model 1.0* states that all application processing happens on the server, including state handling, business logic and page rendering. In addition, the data is persisted with a relational database. *Reference-model 2.0* states that these concerns are now completely implemented on the client, using JavaScript. In addition, the data is persisted by using suitable NoSQL database technologies.

The principles for *Reference-model 1.0* and *Reference-model 2.0* were applied in the implementation of two different architectures that solve the same problem domain; a Web app called Shredhub. These two architectures were respectively called *Architecture 1.0* and *Architecture 2.0*. In order to solve the problem statement, a set of five extensive test cases were designed and performed on these architectures.

The results showed that rendering HTML on the client is fully feasible, by fetching all the necessary HTML to the browser on the initial Web app request. Client-side JavaScript code is especially customized to render only the HTML that is needed for every subsequent request. This resulted in less page rendering and fewer server requests, which successfully lead to less load on the server, and thus both faster response-times and better

scalability was achieved for Architecture 2.0. Better scalability on back-end was also achieved because the amount of work done on the server per request is very limited. A problem however, was that the initial page load was very slow, because a large JavaScript application had to be fetched from the server.

One obvious programming benefit came clear for Architecture 2.0. It was easier to separate the user-interface logic out of the HTML templates, because this logic is implemented in the same language as the business-logic. Therefore, the user-interface logic and business logic that concerns the same domain, could be implemented under the same module, and thus cooperate better.

The tests also showed that moving state and business logic to the client had large performance and scalability gains. The front-end could avoid consulting the server, and when it did, the server merely had to query the database and send small-sized fine-grained data objects back. The results were that the front-end had very quick response times for user actions, and much work was relieved from the server. This led to higher scalability, where many more concurrent users could be served than in the other approach. A pitfall here is that Architecture 2.0 does require an efficient browser and client machine to run efficiently. Another disadvantage is that some business rules has to be implemented both on the client and the server. However, we proposed a hybrid solution for a future study, where when building pure JavaScript Web apps, the code-base might be able to be shared on both the client and the server, thus avoiding the need to duplicate.

As for the databases, both SQL and NoSQL had their advantages. SQL were slow in cases where many join operations were needed. MongoDB could be implemented in a way that avoids the need to join, by having fat objects that contain all the necessary data most queries require. However, once multiple objects had to be joined together, MongoDB could be significantly slow. Another advantage with using MongoDB in Architecture 2.0 was that no marshalling of data was needed to communicate with the database. This is a big programming benefit, because it reduces the size of the code base, and simplifies working with the database. In addition, the flexible NoSQL database allows for the domain to be persisted exactly as it looks like in the user-interface. This very much simplifies query implementations. We also saw that Redis could be used to support the stateless server, by quickly authenticating every HTTP request. Architecture 1.0 uses server-side sessions to maintain authentication through an HTTP session, something that can lead to slow response times because the sessions consumes a lot of memory. Avoiding this with Redis had great performance results for Architecture 2.0, and it facilitates replicating the back-end because the client and server is now much more decoupled.

Part IV

Appendix

Chapter 10

Appendix A

10.1 Architecture 1.0 - Presentation Layer Example

In this section we will see what happens in the controller when a Url request comes in. An example controller, and a controller handler is given in the code below. Extra comments are added to explain important things. Each controller has references to the data source layer, through various service pointers (annotated with `@Autowired`). Also, each controller class maps to a coarse grained URL that references a particular domain, and each controller handler maps to a fine-grained URL that references a particular operation on that domain. The example shows how the follow-shredder action is implemented in the presentation layer.

```
// Handles all Url requests for www.shredhub.com/shredder*
@RequestMapping("/shredder")
@Controller
public class ShredderController {

    // Entrance to the domain logic layer
    @Autowired
    private ShredderService shredderService;

    // Handles a url requests for www.shredhub.com/shredder/<
    // someFaneeId>/?action=follow
    @RequestMapping(value = "/{faneeId}", method = RequestMethod.
        POST, params = "action=follow")
    public String followShredder(@PathVariable int faneeId, Model
        model, HttpSession session) {
        Shredder user = (Shredder) session.getAttribute("user");
        List <Shredder> shreddersFanees = (List <Shredder>) session.
            getAttribute("fanees");
        try {

            // Delegate to the business operation
            List <Shredder> updatedFaneesList = shredderService.
                createFaneeRelation(user(), faneeId, shreddersFanees);

            // Update the session
            session.setAttribute("fanees", updatedFaneesList);
```

```

        // Return the view that displays a list of 20 Shredders
        return this.getShreddersAndReturnShreddersView(model,
            session);

    } catch (IllegalShredderArgumentException e) {
        // Something wrong happened in the business operation.
        // Return the error-page view
        model.addAttribute("errorMsg", e.getMessage());
        return "errorPage";
    }
}

```

If the business operation that was called succeeded, the controller calls a function that does the following:

1. Set p = the current page number stored in the HttpSession object
2. Ask the logic layer to fetch a list of 20 Shredders, starting from page num = p
3. Put the result list on the Model : `model.addAttribute("shreds", resultList)`
4. return the String "shredders"

The returning String is picked up by SpringMVC's View Resolver which is configured to map a Java String to an associated JSP file. Thus the View Resolver will look for a view names "shredders.jsp", or "errorPage.jsp" in case the error view is returned from the handler, and it will render this JSP page together with the model object, resulting in an HTML page that is sent back to the client user.

10.2 Architecture 1.0 - View Logic Example

This example shows how Architecture 1.0 uses three languages to implement view logic; HTML, JSP and JavaScript. The example shows the a simplified version of how a Shred-video is displayed (called a Shred modal). If a user rates or comments the Shred, the new result (I.e updated rating value or the new comment) has to be displayed very quickly in order to achieve proper responsive behavior. This could be solved the usual way by using form-submits, but in this case, this is not good enough, because it results in a complete page refresh in the browser. Everywhere else in the app however, regular form submits are ok.

```

<script type='text/javascript'>
    function commentShred(shredId, commentText) {
        // Create the url that calls the controller handler on the
        // server
        var baseUrl = "<c:url value='/shred/'/>" + shredId;
        var url = baseUrl + "/comment/?text=" + commentText;

        // Send the Ajax request to the server as a Http post
        // request.
    }

```

```

// The result from the server is the Shred with the list of
// comments
// updated with the new comment
$.post(url ,

// This function is called when the server's response comes
// back
function(shred) {
    // get the last comment from the shred, I.e the one the
    // User just created
    var lastComment = _.last(shred.shredComments);

    // Create a comment as an html string ,
    // that is to be injected into the DOM tree
    var htmlString = '<tr><td>' + lastComment.text +
        '</td><td>' + lastComment.commenter.username +
        '</td><td>' + new Date(lastComment.timeCreated).
            toUTCString() + '</td>' +
        '<td><button type="button" class="close"
            onClick="deleteComment(' + last.id + ', ' + data.id + ');
            ">x</button></td></tr>'

    // Append the Html to the table of comments
    $('#commentTable tbody').append(htmlString);
});
}

function rateShred(shredId , commentText) {
    // I have omitted the source code for this example,
    // But it's very similar to the function above
}
</script>

<div class="videoView">
    <video id="videoInModal" src="c:url value="/resources/videos
        /"/>' '$ {shred.videoPath}' </video>
    <p>Created at: ${shred.timeCreated} </p>
    <p>Number of raters:${shred.rating.numberofRaters}</p>
    <p>Rating:${shred.rating.rating}</p>

    <p>Rate it:
        <input type="range" id="rateValue" min="0" max="10" name="
            rating" value="5">
        <button id="rateButton"
            onclick="rateShred($('#rateValue').val()); return false;">
            Rate</button>

    <p>Write a comment</p>
    <input type="text">
    <button id="commentButton"
        onclick="commentShred($('#shredCommentText').val());
        return false;">
        Comment</button>

    <h3>Comments</h3>
    <table id="commentTable">
    <thead>
    <tr>
    <th>Text</th>

```

```

        <th>By</th>
        <th>At</th>
    </tr>
</thead>
<tbody>
    <c:forEach items="${currShred.shredComments}" var="c">
        <tr>
            <td> ${c.text} </td>
            <td> ${c.commenter.username} </td>
            <td> ${c.timeCreated} </td>
        </tr>
    </c:forEach>
</tbody>
</table>
</div>

```

10.3 Architecture 1.0 - Domain Logic Layer Example

This example shows a typical business operation implemented in the domain logic layer. This function is called by the controller handler that receives requests for adding a new shred rating. At first, it tries to fetch the Shred from the database, before it performs some input validation on the data it received. If all went well, the service function will set the new rating and persist the result back to the database. Then it will fetch the Shredder who initially created the Shred in order to increase the Shredder's experience points. Finally it will persist the Shredder back to the database.

```

@Service
@Transactional( readOnly=true )
public class ShredServiceImpl implements ShredService {

    // Data source reference
    @Autowired
    private ShredDAO shredDAO;

    /**
     * This adds a new rating to a Shred.
     * When a shred is rated, the shred will gain a higher total
     * rating, and
     * the shredder who made the shred also achieves
     * more experience points. Note that I don't check if the one
     * who rates the
     * Shred is the one who created it. This is a business rule
     * that should be implemented
     * here, inside the business operation. However, for simplicity
     * I have avoided it.
     */
    @Transactional ( readOnly = false )
    public void rateShred(int shredId, int newRate) throws
        IllegalShredArgumentException {
        Shred shred = shredDAO.getShredById(shredId);
        if ( shred == null ) {
            throw new IllegalShredArgumentException("Shred with id: "
                + shredId + " does not exist");
        }
    }
}

```

```

    if ( newRate < 0 || newRate > 10 ) {
        throw new IllegalArgumentException("Illegal rate
            value!");
    }

    // Here I could use the domain model pattern so that the
    // shred object itself knows how to set its own rating.
    // But I choose to follow the service layer pattern, where
    // all the logic
    // is implemented in this service operation
    ShredRating currentRating = shred.getRating();
    currentRating.setNumberOfRaters(currentRating.
        getNumberOfRaters() + 1);
    currentRating.setCurrentRating(currentRating.
        getCurrentRating()+newRate);

    // store the result in the database
    shredDAO.persistRate(shredId, currentRating);

    // Fetch the complete owner (Shredder) object from the
    // database
    Shredder shredder = shredderDAO.getShredderById(shred.
        getOwner().getId());

    // Uses a utility class that is shared by all the service
    // classes
    // in order to update the shredder level
    UpdateShredderLevel usl = new UpdateShredderLevel(shredder,
        newRate);
    usl.advanceXp();

    shredderDAO.persistShredder( shredder);
}
}

```

10.4 Architecture 1.0 - Add Dig User-feature

In this section, we look at how the Add dig user-feature was implemented. The example starts with the JSP page that visualizes the feature, and the JavaScript that handled the interactive behavior. Note that these are JavaScript functions that contains no modules or namespace declaration. They are just a set of tightly-coupled functions. After this the example shows the implementation on the back-end.

```

// In the Shredder.jsp
<div class="container fullWidth">
    <hr class="soften">

    <h2>Guitar Showroom</h2>
    <br>

    <div class="row-fluid">
        <ul class="thumbnails">

            <li class="span1"></li>

```

```

<li class="span1 arrow-img">
<a href="" data-bypass="true" class="prevImage">
  <i class="icon-backward"></i>
</a></li>

<li class="span8">
  <div class="listContainer">
    <div id="list">
      <c:forEach var="guitar" items="${currentShredder.guitars}">

        <c:if test="${guitar.imgPath != null}">
          <div class="thumbnail">
            ${guitar.imgPath}"
              class="dragImage">
              <p>
                <small>Drag image to see next</small>
              </p>
              <h3>${guitar.name}</h3>
              <p>Digs: ${guitar.diggs} </p>
              <form method="POST"
                action="<c:url value='/' />shredder/${currentShredder.id}/guitar/${guitar.name}/dig">
                <button class="btn btn-success">Dig it</button>
              </form>
            </div>
          </c:if>
        </c:forEach>
      </div>
    </li>

    <li class="span1"></li>
    <li class="span1 arrow-img"><a href="" data-bypass="true"
      class="nextImage"> <i
        class="icon-forward"></i>
      </a></li>
  </ul>
</div>
</div>

<script type="text/javascript">
  $(function() {

    var movedRight = false;
    var mouselsDown = false;
    var slideWidth = 610;
    var slideNumber = 0;
    var xCord = 0;

    $(' .prevImage').on('click', function(event) {
      event.preventDefault();
      slideLeftOrRight(-1);
    });

    $(' .nextImage').on('click', function(event) {
      event.preventDefault();

```

```

        slideLeftOrRight(1);
    });

$('.dragImage').on('mousemove', function(event) {
    event.preventDefault();
    var currXcord = event.pageX;
    if (mouseIsDown) {
        if ( (xCord - currXcord) > 40 ) {
            movedRight=true;
        } else if ( (xCord - currXcord) < 40 ) {
            movedLeft=true;
        }
    }
});

$('.dragImage').on('mousedown', function(event) {
    event.preventDefault();
    mouseIsDown = true;
    xCord = event.pageX;
});

$('.dragImage').on('mouseup', function(event) {
    mouseIsDown = false;
    if ( movedRight == true ) {
        movedRight = false;
        slideLeftOrRight(1);
    } else if ( movedLeft == true ) {
        movedLeft = false;
        slideLeftOrRight(-1);
    }
});

function slideLeftOrRight(step) {
    slideNumber += step; // 1 / 2 / 3 / 4 / ... n
    var containingUL = document.getElementById("list");
    slideTo(containingUL, -slideNumber * slideWidth);
}

function slideTo(el, left) {
    var steps = 10;
    var timer = 25;
    var elLeft = parseInt(el.style.left) || 0;
    var diff = left - elLeft;
    var stepSize = diff / steps;

    function step() {
        elLeft += stepSize;
        el.style.left = elLeft + "px";
        if (--steps) {
            setTimeout(step, timer);
        }
    }
    step();
}

function changeImg(event, step) {
    event.preventDefault();

```

```

        slideNumber += step;
        var containingUL = document.getElementById("list");
        slideTo(containingUL, -slideNumber * slideWidth);
    }
    });
</script>

// In ShredderController
@RequestMapping( value =("/{id}/guitar/{guitaName}/dig", method =
    RequestMethod.POST)
public String digGuitar(@PathVariable String id, @PathVariable
    String guitName,
    HttpSession session, Model model) {
    Shredder shredder = (Shredder) session.getAttribute("shredder");
    boolean res = shredderService.addDiggForGuitar(shredder, id,
        guitName);
    if ( res ) {
        return "redirect:/shredder/" + id;
    } else {
        return "errorPage"; // Add error message
    }
}

// In ShredderService
@Transactional(readOnly=false)
public boolean addDiggForGuitar(Shredder user, String id, String
    gIndex) {
    if ( !(user.getId() + "").equals(id) ){

        boolean sqlRes = shredderDAO.addDiggForGuitar(id, gIndex);
        if ( sqlRes ) {
            Shredder shredder = shredderDAO.getShredderById(Integer.
                parseInt(id));

            // Update shredder level
            UpdateShredderLevel usl = new UpdateShredderLevel(shredder,
                1 );
            boolean res = usl.advanceXp();
            shredderDAO.persistShredder( shredder);
            return true;
        }
    }
    return false;
}

// In UpdateShredderLevel
public UpdateShredderLevel(Shredder sh, int newPoints) {
    this.shredder = sh;
    this.newPoints = newPoints;
}

/**
 * @return true if leveled up, false otherwise
 */
public boolean advanceXp() {

```



```

    ShredderLevel sl = shredder.getLevel();
    if ( shouldLevelUp(sl.getXp()) ) {
        doLevelUp(sl);
        return true;
    } else {
        sl.setXp( sl.getXp() + newPoints);
        return false;
    }
}

private void doLevelUp(ShredderLevel sl) {
    sl.setLevel( sl.getLevel() + 1);
    sl.setXp((sl.getXp() + newPoints) % POINTS_FOR_NEW_LEVEL);
}

private boolean shouldLevelUp(double xp) {
    return (( xp + newPoints)) >= 100;
}

// In ShredderDAOImpl
public boolean addDiggForGuitar(String id, String guitarName) {
    int shredderId = Integer.parseInt(id);
    String SQL = "UPDATE GuitarForShredder SET Digs = Digs+? WHERE
        ShredderId=? AND Guitar=?";
    try{
        jdbcTemplate.update(SQL, 1, shredderId, guitarName);
        return true;
    } catch (DataAccessException e) {
        return false;
    }
}

public void persistShredder(Shredder shredder) {
    jdbcTemplate.update("UPDATE Shredder SET Username=?, Birthdate
        =?,Email=?,Password=?" +
        ",Description=?,Country=?,ProfileImage=?,ExperiencePoints=?,
        ShredderLevel=?" +
        "WHERE Id=?", shredder.getUsername(), shredder.getBirthdate
        (), shredder.getEmail(),
        shredder.getPassword(), shredder.getDescription(), shredder.
        getCountry(), shredder.getProfileImagePath(),
        shredder.getLevel().getXp(), shredder.getLevel().getLevel(),
        shredder.getId());
}

```


Chapter 11

Appendix B

11.1 Architecture 2.0 - Domain Logic Example

The following example shows the business logic implemented for the rate-shred user action. The example shows the implementation in the Shred and Shredder Model objects, and the associated API function. Note that the API functions also contain some business logic in order to verify correct input data. There are two API calls performed in this example. The first updates the rating on a Shred. The other updates the shredder level on the Shredder who owns the Shred. The model functions are called by a View implementation (showed in the next section).

```
// In the Shred Model module
addRating : function(rateValue) {
  if( rateValue <0 || rateValue > 10){
    return false;
  }
  var shredRating = this.get('shredRating');
  shredRating.numberOfRaters ++;
  shredRating.currentRating += parseInt(rateValue, 10);

  // Call the Shredder module.
  this.increaseShredderLevel(parseInt(rateValue,10));

  // Send the updated JSON object to the API
  // calls updateShred in the API
  this.save();

  // Notify the View of the change so it can update the UI
  this.trigger('change');
  return true;
},

increaseShredderLevel : function(level) {
  var shredder = new Shredder.Model(this.get('owner'));
  return shredder.increaseShredderLevel(level);
},
});

// In the Shredder Model module
increaseShredderLevel : function(level) {
```

```

        var that = this;
        // No need to evaluate the rate value. It's already been
        // verified!

    // Calls updateShredder in the API
    $.ajax(this.urlRoot + "/" + this.get('_id'),
    {
        // add Basic http authentication header info
        beforeSend : function(xhr) {
            xhr.setRequestHeader("Authorization", ("Basic " +
            concat(Session.getToken())));
        },
        data : { shredderLevel : level },
        type : "PUT"
    })

    // Upon success, update the Shredder object, and notify the
    // UI
    // of the change
    .done(function(res){
        that.set(res);
        that.trigger('shredderUpdated');
    });
    return true;
},

// In the API, Shred repository
// @req is the http request object
// @res is the http response object
exports.updateShred = function(req,res) {
    // check if the owner is not the same as the current User
    if ( req.user.uid == req.body.owner._id){
        res.statusCode = 401;
        return res.send(null);
    }

    // get the shred object from MongoDB so we can add some business
    // rules
    // check that the new rating is less than 10, more than 0
    shred.getShred(req.body._id)
    .done(function(leShred){

        var newRateVal = req.body.shredRating.currentRating -
            leShred.shredRating.currentRating;
        if ( newRateVal < 0 || newRateVal > 10){
            res.statusCode = 400;
            return res.send(null);
        }

        // Everything ok. Update the JSON object in MongoDB shred
        // wrapper
        shred.updateShredByObject(req.body)
        .done(function(doc){
            res.send(doc);
        })
        .fail(function(err){
            res.send(null);
        })
    })

```

```

    });
}

// In the API, Shredder repository
exports.updateShredder = function(req,res) {

    // Current user is not allowed to update his own shredder level
    if ( req.user.uid == req.params.uid){
        res.statusCode = 401;
        return res.send(null);
    }

    // Check shredder level is within bounds
    var newLevel = req.body.shredderLevel;
    if ( newLevel > 10 || newLevel < 0) {
        res.statusCode = 400;
        return res.send(null);
    }

    // call the update function on MongoDB shredder wrapper
    shredder.updateShredder(req.params.uid, req.body)
    .done(function(doc){
        res.send(doc);
    })
    .fail(function() {
        res.send(null);
    })
}

```

11.2 Architecture 2.0 - View Logic Example

In this example, we see how the view logic is implemented in Architecture 2.0. Each view object implements all the view logic necessary for that particular view. They cooperate closely with the model to delegate business operations. Each View has a reference to the HTML template they “own”. This example shows the view logic for displaying a Shred-video.

```

Shred.Views.ModalView = BaseView.extend({
    template: "shred/shredModal",

    /** INITIALIZATION CODE */

    // After the rendering process is done,
    //add the necessary event handlers
    postRender : function() {
        $('#rateButton').on("click", $.proxy(this.rateShred, this))
        ;
        $('#commentButton').on("click", $.proxy(this.saveComment,
            this));
        $('td.close').on("click", $.proxy(this.deleteComment, this
        ));
        this.listenTo(this.model, 'change', this.notifyOnChange);
    },

    // Return a JSON object that will be rendered into the HTML
    serialize : function() {

```

```

        return {"shred" : this.model.toJSON() };
    },

    /* EVENT HANDLERS */

    // Rate button clicked
    rateShred : function(event) {
        event.preventDefault();
        var rateVal = $('input[type=range]').val();

        // Call the business operation in the model
        this.model.addRating(rateVal);
    },

    // Delete comment clicked. Call the business function in the
    // model
    deleteComment : function(event) {
        var commentIndex = event.currentTarget.id.split("-")[1];
        this.model.deleteComment(commentIndex);
    },

    // A new comment is created. Call the model function for
    // business op!
    saveComment : function(event) {
        event.preventDefault();
        this.model.addComment($('input[id=#commentText]').val(), Session.
            getUser());
    },

    /** CLEAN UP FUNCTIONS */

    // Called when the shred object has been updated. Must re-
    // render the view
    notifyOnChange : function() {
        app.Mediator.publish("createShredModalView", this.model);
    },

    // Called when a this view is to be re-used to display a new
    // Shred
    resetShredModel : function(newModel) {
        this.stopListening(this.model, 'change', this.notifyOnChange
        );
        this.model = newModel;
    },

    // Kill this view by deleting its DOM elements and remove
    // event handlers
    cleanUp : function() {
        console.log("Killing shredmodal " + this.cid);
        this.remove();
        this.unbind();
    }
});

```

11.3 Architecture 2.0 - Add Dig User-feature

In this section, we look at how the Add dig user-feature was implemented. The example starts with the view implementation that concerns dragging a guitar image to the left or right in order to scroll through the list, and clicking on the add dig button on one of the guitars. The example continues with the business operation in the Shredder Model, and finally the API implementation. Once again, note that some business logic is duplicated on the back-end in order to prevent malformed input.

```
// In the Shredder template
<div class="container fullWidth">
  <h2>Guitar Showroom</h2>

  <div class="row-fluid">
    <ul class="thumbnails">

      <li class="span1"></li>
      <li class="span1 arrow-img">
        <a href="" data-bypass="true" class="guitarLeftClick">
          <i class="icon-backward" ></i>
        </a>
      </li>

      <li class="span8">
        <div class="listContainer">
          <div id="list">
            <% _.each(shredder.guitars , function(guitar) { %>

              <% if ( ( typeof guitar ) == 'object' ) { %>
                <div class="thumbnail">
                  
                  <p><small>Drag image to see next</small></p>
                  <h3><%= guitar.type %></h3>
                  <p>Diggs: <%= guitar.digs %> </p>
                  <p><button class="btn btn-success digBtn">Dig</
                    button></p>
                </div>
              <% } }) %>
            </div>
          </div>
        </li>

        <li class="span1"></li>
        <li class="span1 arrow-img">
          <a href="" data-bypass="true" class="guitarRightClick">
            <i class="icon-forward"></i>
          </a>
        </li>
      </ul>
    </div>

  </div>
```

```
// In the Shredder View
```

```

// Reference to the Html template
template : "shredder/Shredder",

// Re-set some state data
initialize : function() {
    this.slideWidth = 610;
    this.slideNumber = 0;

    // Update the UI when the shredder is updated
    this.listenTo(this.model, 'shredderUpdated', this.renderTemplate);
},

events : {
    "mousedown .dragImage" : "mouseDownEvent" ,
    "mousemove .dragImage" : "mouseMoveEvent" ,
    "mouseup .dragImage" : "mouseUpEvent" ,
    "click .guitarRightClick" : "rightClick" ,
    "click .guitarLeftClick" : "leftClick" ,
    "click .digBtn" : "digButton"
},

// Dig button is clicked. Add a digg in the Model object
digButton : function() {
    this.model.addDig(this.slideNumber);
},

rightClick: function(event){
    event.preventDefault();
    this.slideLeftOrRight(1);
},

leftClick: function(event){
    event.preventDefault();
    this.slideLeftOrRight(-1);
},

slideLeftOrRight : function(step) {
    this.slideNumber += step; // 1 / 2 / 3 / 4 / ... n
    var containingUL = document.getElementById("list");
    this.slideTo(containingUL, -this.slideNumber * this.slideWidth);
},

slideTo : function(el, left) {
    var steps = 10;
    var timer = 25;
    var elLeft = parseInt(el.style.left, 10) || 0;
    var diff = left - elLeft;
    var stepSize = diff / steps;

    function step() {
        elLeft += stepSize;
        el.style.left = elLeft + "px";
        if (--steps) {
            setTimeout(step, timer);
        }
    }
}

```



```

    }
  }
  step();
},

mouseUpEvent : function(event) {
  this.mousIsDown = false;
  if ( this.movedRight === true ) {
    this.movedRight = false;
    this.slideLeftOrRight(1);
  } else if ( this.movedLeft === true ) {
    this.movedLeft = false;
    this.slideLeftOrRight(-1);
  }
},

mouseDownEvent : function(event) {
  event.preventDefault();
  this.mousIsDown = true;
  this.xCord = event.pageX;
},

mouseMoveEvent : function(event) {
  event.preventDefault();
  var currXcord = event.pageX;
  if ( this.mousIsDown ) {
    if ( (this.xCord - currXcord) > 40 ) {
      this.movedRight=true;
    } else if ( (this.xCord - currXcord) < 40 ) {
      this.movedLeft=true;
    }
  }
},

// In the Shredder Model
addDig : function(i) {
  var that = this;

  // The user cannot add dig to his own guitar
  if (this.get('_id') != Session.getUser()._id) {

    // Call the API
    $.ajax(this.urlRoot + '/' + this.get('_id') + '/guitar/' +
      i + '/dig',
      {

        // Add authentication header
        beforeSend : function(xhr) {
          xhr.setRequestHeader("Authorization",
            ("Basic '".concat(Session.getToken())));
        },
        type : 'PUT'
      })
    .done(function(res) {

      // Increase the shredder level for this shredder
      // See the previous section for the implementation of
      this
    })
  }
}

```

```

        that.increaseShredderLevel(1);
    });
    } else {
        return {errorMsg : "Cannot add dig to your own guitar"};
    }
},

// in the API
exports.digGuitar = function(req,res) {

    // check if the owner is not the same as the current User
    if ( req.user.uid == req.body.owner._id){
        res.statusCode = 401;
        return res.send(null);
    }

    return shredder.addDigForGuitar({
        uid: req.params.uid,
        gIndex : req.params.gIndex,
        res : res
    });
}

// In the MongoDB client
exports.addDigForGuitar = function(arg) {
    var guitarArr ={}
    guitarArr["guitars." + arg.gIndex + ".digs"] = 1;

    exports.Shredder.update({"_id" : arg.uid.toString()}, {$inc :
        guitarArr},
        function(err, numberAffected, raw) {return raw;});
}

```

Bibliography

- [1] *A Little History of the World Wide Web*. URL: <http://www.w3.org/History.html>.
- [2] Daniel J Abadi. 'Data Management in the Cloud: Limitations and Opportunities'. In: *IEEE* (2009).
- [3] *ActiveX*. URL: <http://support.microsoft.com/kb/154544>.
- [4] *Adobe Flash*. URL: <http://www.adobe.com/products/flashruntimes.html>.
- [5] *Airbnb*. URL: <http://www.airbnb.com/>.
- [6] Brown et al. *Enterprise Java Programming with IBM WebSphere*. Addison-Wesley, 2001.
- [7] *Apache*. URL: <http://httpd.apache.org/>.
- [8] *Apache CouchDB*. URL: <http://couchdb.apache.org/>.
- [9] *Apache JMeter*. URL: <http://jmeter.apache.org/>.
- [10] *Apache Tomcat*. URL: <http://tomcat.apache.org/>.
- [11] Gaurav Asthana. *MongoDB vs MySQL: speed test part 1, Insert queries*. URL: <http://gauravasthana.blogspot.no/2011/01/mongodb-vs-mysql-speed-test-part-1.html>.
- [12] Gaurav Asthana. *MongoDB vs MySQL: speed test part 2, Select queries*. URL: <http://gauravasthana.blogspot.no/2011/01/mongodb-vs-mysql-speed-test-part-2.html>.
- [13] *Asynchronous module definition*. URL: http://en.wikipedia.org/wiki/Asynchronous_module_definition.
- [14] *Backbone.js*. URL: <http://backbonejs.org/>.
- [15] Sumita Barahmand and Shahram Ghandeharizadeh. 'BG: A Benchmark to Evaluate Interactive Social Networking Actions'. In: *CoRR, Proceedings of 2013 CIDR* (2012).
- [16] Spike Brehm. *Our First Node.js App: Backbone on the Client and Server*. URL: <http://nerds.airbnb.com/weve-launched-our-first-nodejs-app-to-product>.
- [17] Severance C. 'JavaScript: Designing a Language in 10 Days'. In: *IEEE Computer Society*, (Vol. 45, No. 2) pp. 7-8 (2012).
- [18] *Cassandra*. URL: <http://cassandra.apache.org/>.

- [19] Rick Cattell. 'Scalable SQL and NoSQL data stores'. In: *ACM SIGMOD Record* 39.4 (2011), pp. 12–27.
- [20] Steve Champeon. *JavaScript: How Did We Get Here?* URL: http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html.
- [21] Fay Chang et al. 'Bigtable: A distributed storage system for structured data'. In: *ACM Transactions on Computer Systems* 26.2: 4 (2008).
- [22] Stephen Chapman. *A Brief History of Javascript*. URL: <http://javascript.about.com/od/reference/a/history.htm>.
- [23] Henrik B. Christensen. *Flexible, Reliable Software: Using Patterns and Agile Development*. Taylor & Francis, 2011.
- [24] *Chrome Browser*. URL: <http://www.google.com/intl/com/chrome/browser/>.
- [25] *Chrome DevTools*. URL: <https://developers.google.com/chrome-developer-tools/>.
- [26] *ClojureScript*. URL: <https://github.com/clojure/clojurescript>.
- [27] E. F Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.
- [28] *CoffeScript*. URL: <http://coffeescript.org/>.
- [29] *Common Gateway Interface*. URL: <http://web.archive.org/web/20100127191128/http://hoohoo.ncsa.illinois.edu/cgi/intro.html>.
- [30] Dave Crane and Phil McCarthy. 'Comet and Reverse Ajax: The Next-Generation Ajax 2.0'. In: *Apress* (2008).
- [31] Douglas Crockford. *JavaScript: The World's Most Misunderstood Programming Language*. URL: <http://www.crockford.com/javascript/javascript.html>.
- [32] *CSS*. URL: <http://www.w3.org/Style/CSS/>.
- [33] *DB-Engines Ranking*. URL: <http://db-engines.com/en/ranking>.
- [34] *Django*. URL: <https://www.djangoproject.com/>.
- [35] *Dojo*. URL: <http://dojotoolkit.org/>.
- [36] Thomas E. 'Service-oriented architecture'. In: *Prentice Hall* (2004).
- [37] *Enterprise Apps Are Moving To Single-Page Design*. URL: <http://techcrunch.com/2012/11/30/why-enterprise-apps-are-moving-to-single-page-design/>.
- [38] *Extensible Markup Language (XML)*. URL: <http://www.w3.org/XML/>.
- [39] *Facebook*. URL: <http://www.facebook.com>.
- [40] Roy Thomas Fielding. 'Architectural Styles and the Design of Network-based Software Architectures'. PhD thesis. University of California, Irvine, 2000.
- [41] Martin Fowler. *Front Controller Pattern*. URL: <http://www.martinfowler.com/eaCatalog/frontController.html>.

- [42] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. URL: <http://martinfowler.com/articles/injection.html>.
- [43] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.
- [44] Armando Fox and David Patterson. *Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing*. Strawberry Canyon LLC.
- [45] *Framework Benchmarks*. URL: <http://www.techempower.com/blog/2013/03/28/framework-benchmarks/>.
- [46] John Franks et al. *HTTP authentication: Basic and digest access authentication*. RFC 2617, June. 1999.
- [47] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
- [48] Jesse James Garrett. *Ajax: A New Approach to Web Applications*. URL: <http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications>.
- [49] *Gecko*. URL: <https://developer.mozilla.org/en-US/docs/Mozilla/Gecko>.
- [50] *GitHub*. URL: <https://github.com/>.
- [51] *Google*. URL: <https://www.google.com/intl/us/about/>.
- [52] *Google AdSense*. URL: <http://www.google.com/adsense>.
- [53] *Google Analytics*. URL: <http://www.google.com/analytics>.
- [54] *Google Gmail*. URL: <https://mail.google.com/>.
- [55] *Google Maps*. URL: <https://maps.google.com/>.
- [56] William G.J. Halfond, Jeremy Viegas and Alessandro Orso. 'A Classification of SQL Injection Attacks and Countermeasures'. In: (2006).
- [57] *HamL*. URL: <http://haml.info/>.
- [58] *Hibernate*. URL: <http://www.hibernate.org/>.
- [59] *Hot Frameworks*. URL: <http://www.hotframeworks.com/rankings>.
- [60] *HTML5, The History interface*. URL: <http://www.whatwg.org/specs/web-apps/current-work/multipage/history.html#the-history-interface>.
- [61] *HTML5, Web Storage*. URL: <http://www.w3.org/TR/webstorage/>.
- [62] *Internet Explorer*. URL: <http://windows.microsoft.com/en-us/internet-explorer/download-ie>.
- [63] *Internet Information Services (IIS)*. URL: <http://www.iis.net/>.
- [64] *Java Platform, Enterprise Edition*. URL: <http://www.oracle.com/technetwork/java/javaee/overview/index.html>.
- [65] *JavaScript Leads The Pack As Most Popular Programming Language*. URL: <http://www.webpronews.com/javascript-leads-the-pack-as-most-popular-programming-language-2012-09>.

- [66] *JavaScript Web Applications*. O'Reilly Media, 2011.
- [67] *JavaServer Pages Technology*. URL: [http : / / www . oracle . com / technetwork/java/jsp-138432.html](http://www.oracle.com/technetwork/java/jsp-138432.html).
- [68] *jQuery*. URL: <http://www.jquery.com/>.
- [69] *JScript (ECMAScript3)*. URL: [http : / / msdn . microsoft . com / en - us / library/hbxc2t98\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/hbxc2t98(v=vs.85).aspx).
- [70] *JSON*. URL: <http://www.json.org/>.
- [71] Eric Lai. *No to SQL? Anti-database movement gains steam*. URL: [http : / / www . computerworld . com / s / article / 9135086 / No _ to _ SQL _ Anti _ database _ movement _ gains _ steam _](http://www.computerworld.com/s/article/9135086/No_to_SQL_Anti_database_movement_gains_steam_).
- [72] *LINQ (Language-Integrated Query)*. URL: <http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx>.
- [73] *List of NoSQL Databases*. URL: <http://nosql-database.org/>.
- [74] Andrew Malkov. *JavaScript modularity with RequireJS (from spaghetti code to ravioli code)*. URL: [http : / / netmvc . blogspot . no / 2012 / 11 / javascript-modularity-with-requirejs.html](http://netmvc.blogspot.no/2012/11/javascript-modularity-with-requirejs.html).
- [75] P. Mazzetti, S. Nativi and L. Bigagli. 'Integration of REST style and AJAX technologies to build Web applications; an example of framework for Location-Based-Services'. In: *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*. 2008, pp. 1–6.
- [76] *Memcached*. URL: <http://memcached.org/>.
- [77] Ali Mesbah and Arie van Deursen. 'Migrating multi-page web applications to single-page Ajax interfaces'. In: *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*. IEEE. 2007, pp. 181–190.
- [78] *Microsoft Application Architecture Guide, 2nd Edition*. URL: [http : / / msdn . microsoft . com / en - us / library/ee658124.aspx](http://msdn.microsoft.com/en-us/library/ee658124.aspx).
- [79] *Microsoft Silverlight*. URL: <http://www.microsoft.com/silverlight/>.
- [80] *Microsoft SQL Server*. URL: [http : / / www . microsoft . com / en - us / sqlserver/default.aspx](http://www.microsoft.com/en-us/sqlserver/default.aspx).
- [81] Michael S. Mikowski and Josh C. Powell. *Single Page Web Applications*. Manning Publications Co, 2013.
- [82] *mongoDB*. URL: <http://www.mongodb.org/>.
- [83] *MongoDB – The Leading NoSQL Database*. URL: [http : / / www . 10gen . com / leading - nosql - database](http://www.10gen.com/leading-nosql-database).
- [84] *Mosaic*. URL: <http://www.totic.org/nscp/demodoc/demo.html>.
- [85] *Mozilla*. URL: <http://www.mozilla.org/en-US/>.
- [86] *Mustache, logic-less templates*. URL: <http://mustache.github.io/>.
- [87] *MyBatis*. URL: <https://code.google.com/p/mybatis/>.
- [88] *MySQL*. URL: <http://www.mysql.com/>.

- [89] *New DB-Engines Ranking shows the popularity of database management systems*. URL: http://db-engines.com/en/blog_post/1.
- [90] *Node.js*. URL: <http://nodejs.org/>.
- [91] *OAuth*. URL: <http://oauth.net/>.
- [92] M. Ohara et al. 'The data-centricity of Web 2.0 workloads and its impact on server performance'. In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. 2009, pp. 133–142.
- [93] *Oracle SQL*. URL: <http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html>.
- [94] Tim Oreilly. 'What is Web 2.0: Design patterns and business models for the next generation of software'. In: *Communications & strategies* (2007).
- [95] Addy Osmani. *Learning JavaScript Design Patterns*. URL: <http://addyosmani.com/resources/essentialjsdesignpatterns/book/>.
- [96] *PHP*. URL: <http://php.net/>.
- [97] *Pinterest*. URL: <http://www.pinterest.com>.
- [98] *PlanetLab*. URL: <http://www.planet-lab.org/>.
- [99] *PostgreSQL*. URL: <http://www.postgresql.org/>.
- [100] *PostgreSQL vs MongoDB*. URL: <http://blog.pingoured.fr/index.php?post/2012/05/20/PostgreSQL-vs-MongoDB>.
- [101] Katharina Probst et al. *A proposal for making AJAX crawlable*. URL: <http://googlewebmastercentral.blogspot.no/2009/10/proposal-for-making-ajax-crawlable.html>.
- [102] *Project Voldemort*. URL: <http://www.project-voldemort.com/voldemort/>.
- [103] *Prototype JavaScript framework: a foundation for ambitious web user interfaces*. URL: <http://www.prototypejs.org/>.
- [104] Johnson R. *Expert One on One J2ee Design and Development*. Wrox, 2003.
- [105] *Rack: a Ruby Webserver Interface*. URL: <http://rack.github.io/>.
- [106] *Redis*. URL: <http://www.redis.io/>.
- [107] Trygve Mikjel H Reenskaug. 'The original MVC reports'. In: (1979).
- [108] *Riak*. URL: <http://basho.com/riak/>.
- [109] *Ruby on Rails*. URL: <http://rubyonrails.org/>.
- [110] Jose Maria Arranz Santamaria. *The Single Page Interface Manifesto*. URL: http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php.
- [111] *Scaling Pinterest - From 0 To 10s Of Billions Of Page Views A Month In Two Years*. URL: http://highscalability.com/blog/2013/4/15/scaling-pinterest-from-0-to-10s-of-billions-of-page-views-a.html?utm_source=feedly.

- [112] *Separation of Business Logic from Presentation Logic in Web Applications (ASP.NET and PHP)*. URL: <http://www.paragoncorporation.com/ArticleDetail.aspx?ArticleID=21>.
- [113] Nicolás Serrano and Juan Pablo Aroztegi. 'Ajax frameworks in interactive web apps'. In: *Software, IEEE* (2007).
- [114] *Spaghetti code*. URL: http://en.wikipedia.org/wiki/Spaghetti_code.
- [115] *Spring in Action, Third Edition*. Manning Publications Co, 2011.
- [116] Randy Stafford. *Service Layer*. URL: <http://martinfowler.com/eaCatalog/serviceLayer.html>.
- [117] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education Inc., 2011.
- [118] *The Apache Velocity Project*. URL: <http://velocity.apache.org/>.
- [119] *The Internet Engineering Task Force (IETF)*. URL: <http://www.ietf.org/>.
- [120] *The Internet Explorer 6 Countdown*. URL: <http://www.ie6countdown.com/>.
- [121] *The Java Persistence API*. URL: <http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html>.
- [122] *The WebKit Open Source Project*. URL: <http://www.webkit.org/>.
- [123] *The website of the world's first-ever web server*. URL: <http://info.cern.ch/>.
- [124] *TodoMVC, Helping you select an MV* framework*. URL: <https://github.com/addyosmani/todomvc>.
- [125] *Twitter*. URL: <http://www.Twitter.com>.
- [126] *V8 JavaScript Engine*. URL: <https://code.google.com/p/v8/>.
- [127] Alvaro Videla. <http://obvioushints.blogspot.no/2009/07/benchmarking-mongodb-vs-mysql.html>. URL: <http://obvioushints.blogspot.no/2009/07/benchmarking-mongodb-vs-mysql.html>.
- [128] *Web 2.0 Architectures*. O'Reilly Media, 2009.
- [129] Dan Webb. *Improving performance on twitter.com*. URL: <http://engineering.twitter.com/2012/05/improving-performance-on-twittercom.html>.
- [130] *WebSocket*. URL: <http://www.websocket.org/>.
- [131] Steven Willmott. 'HTML, Javascript and the app-ification of the Web'. June 2012.
- [132] *World Wide Web Consortium (W3C)*. URL: <http://www.w3.org/>.